

CONFERENCE PROCEEDINGS

U S E N I X

W I N T E R 1 9 9 4

C O N F E R E N C E



JAN. 17-21, 1994

SAN FRANCISCO

C A L I F O R N I A

USENIX®

THE UNIX AND ADVANCED COMPUTING
PROFESSIONAL AND TECHNICAL ASSOCIATION

USENIX

SAN FRANCISCO CONFERENCE PROCEEDINGS

WINTER

1994

For additional copies of these proceedings write:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

The price is \$30 for members and \$39 for nonmembers.
Outside the U.S.A. and Canada, please add
\$20 per copy for postage (via air printed matter).

Past USENIX Technical Conferences

1993 Summer Cincinnati	1987 Summer Phoenix
1993 Winter San Diego	1987 Winter Washington, DC
1992 Summer San Antonio	1986 Summer Atlanta
1992 Winter San Francisco	1986 Winter Denver
1991 Summer Nashville	1985 Summer Portland
1991 Winter Dallas	1985 Winter Dallas
1990 Summer Anaheim	1984 Summer Salt Lake City
1990 Winter Washington, DC	1984 Winter Washington, DC
1989 Summer Baltimore	1983 Summer Toronto
1989 Winter San Diego	1983 Winter San Diego
1988 Summer San Francisco	
1988 Winter Dallas	

1994 © Copyright by The USENIX Association
All Rights Reserved.

ISBN 1-880446-58-8

This volume is published as a collective work.
Rights to individual papers remain with the author or the author's employer.

USENIX acknowledges all trademarks herein.

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.



USENIX Association

**Proceedings of the
Winter 1994 USENIX Conference**

**January 17 - 21, 1994
San Francisco, California, USA**

For additional copies of these proceedings write:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

The price is \$30 for members and \$39 for nonmembers.
Outside the U.S.A. and Canada, please add
\$20 per copy for postage (via air printed matter).

Past USENIX Technical Conferences

1993 Summer Cincinnati	1987 Summer Phoenix
1993 Winter San Diego	1987 Winter Washington, DC
1992 Summer San Antonio	1986 Summer Atlanta
1992 Winter San Francisco	1986 Winter Denver
1991 Summer Nashville	1985 Summer Portland
1991 Winter Dallas	1985 Winter Dallas
1990 Summer Anaheim	1984 Summer Salt Lake City
1990 Winter Washington, DC	1984 Winter Washington, DC
1989 Summer Baltimore	1983 Summer Toronto
1989 Winter San Diego	1983 Winter San Diego
1988 Summer San Francisco	
1988 Winter Dallas	

1994 © Copyright by The USENIX Association
All Rights Reserved.

ISBN 1-880446-58-8

This volume is published as a collective work.
Rights to individual papers remain with the author or the author's employer.

USENIX acknowledges all trademarks herein.

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.



TABLE OF CONTENTS

USENIX Winter 1994 Technical Conference
January 17-21, 1994
San Francisco, California

Monday (9:00 - 10:30)

Opening Remarks

Jeffrey Mogul, Digital Equipment Corporation, Western Research Laboratory

Keynote Address

John Perry Barlow, Electronic Frontier Foundation

SIMPLE DATABASE TOOLS

Monday (11:00 - 12:30)

Chair: Rafael Alonso

Finding Similar Files in a Large File System.....	1
<i>Udi Manber, Department of Computer Science, University of Arizona</i>	
cql - A Flat File Database Query Language.....	11
<i>Glenn Fowler, AT&T Bell Laboratories</i>	
GLIMPSE: A Tool to Search Through Entire File Systems	23
<i>Udi Manber, Sun Wu, Department of Computer Science, University of Arizona</i>	

WIDE - AREA INFORMATION ACCESS

Monday (2:00 - 3:30)

Chair: Frederick S. Glover

Drinking from the Firehose: Multicast USENET News	33
<i>Kurt J. Lidl, Josh Osborne, Joe Malcolm, UUNET Technologies, Inc.</i>	
The redbms Distributed Bibliographic Database System.....	47
<i>Richard A. Golding, Vrije Universiteit, Darrell D. E. Long, University of California, Santa Cruz; John Wilkes, Hewlett-Packard Laboratories</i>	
Filesystem Daemons as Unifying Mechanism for Network Information Access.....	63
<i>Steve Summit, Consultant, Seattle, Washington</i>	

INTERPROCESS COMMUNICATION

Monday (4:00 - 5:30)

Chair: Cathy Watkins

Concert/C: A Language for Distributed Programming.....	79
<i>Joshua S. Auerbach, Arthur P. Goldberg, Ajei S. Gopal, Mark T. Kennedy, James R. Russell, IBM T. J. Watson Research Center</i>	
Evolving Mach 3.0 to A Migrating Thread Model	97
<i>Bryan Ford, Jay Lepreau, Department of Computer Science, University of Utah</i>	

Tread Marks: Distributed Shared Memory on Standard Workstations and Operating System	115
<i>Peter Keleher, Alan L. Cox, Sandhya Dwarkadas, Willy Zwaenepoel, Department of Computer Science, Rice University</i>	

EFFICIENT SCHEDULING AND NETWORKING

Tuesday (9:00 - 10:30)

Chair: Michael B. Jones

Workstation Support for Real-Time Multimedia Communication	133
<i>Olof Hagsand, Peter Sjödin, Swedish Institute of Computer Science</i>	
Experience and Results from Implementation of an ATM Socket Family	143
<i>Richard Black, Simon Crosby, University of Cambridge Computer Laboratory</i>	
Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages	153
<i>Masanobu Yuhara, Fujitsu Laboratories Ltd., Brian N. Bershad, Chris Maeda, School of Computer Science, Carnegie Mellon University; J. Eliot B. Moss, Department of Computer Science, University of Massachusetts, Amherst</i>	

KERNEL PERFORMANCE

Tuesday (11:00 - 12:30)

Chair: Brian N. Bershad

Latency Analysis of TCP on an ATM Network	167
<i>Alec Wolman, Geoff Voelker, Chandramohan A. Thekkath, Department of Computer Science and Engineering, University of Washington</i>	
Improving UNIX Kernel and Networking Performance Using Profile Based Optimization	181
<i>Steven E. Speer, Rajiv Kumar, Hewlett-Packard; Craig Partridge, Bolt Beranek and Newman</i>	
Memory Behavior for an X11 Window System.....	189
<i>J. Bradley Chen, School of Computer Science, Carnegie Mellon University</i>	

SYSTEM DEVELOPMENT TOOLS

Tuesday (2:00 - 3:30)

Chair: Judith E. Grass

A Uniform Name Service for Spring's UNIX Environment.....	201
<i>Michael N. Nelson, Silicon Graphics, Inc.; Sanjay R. Radia, SunSoft, Inc.</i>	
ACID: A Debugger Built from a Language.....	211
<i>Phil Winterbottom, AT&T Bell Laboratories</i>	
Acme: A User Interface for Programmers	223
<i>Rob Pike. AT&T Bell Laboratories</i>	

NFS

Wednesday (9:00 - 10:30)

Chair: Margo Seltzer

File System Design for an NFS File Server Appliance	235
<i>Dave Hitz, James Lau, Michael Malcolm, Network Appliance Corporation</i>	

Improving the Write Performance of an NFS Server.....	247
<i>Chet Juszczak, Digital Equipment Corporation</i>	
Not Quite NFS, Soft Cache Consistency for NFS	261
<i>Rick Macklem, Department of Computing and Information Science, University of Guelph</i>	

DISKS AND FILE SYSTEMS

Wednesday (11:00 - 12:30)

Chair: David Presotto

A Quantitative Analysis of Disk Drive Power Management in Portable Computers.....	279
<i>Kester Li, Roger Kumpf, Paul Horton, Thomas Anderson, Computer Science Division, University of California, Berkeley</i>	
Thwarting the Power-Hungry Disk.....	293
<i>Fred Douglass, P. Krishnan, Brian Marsh, Matsushita Information Technology Laboratory</i>	
A Usage Profile and Evaluation of a Wide-Area Distributed File System	307
<i>Mirjana Spasojevic, Transarc Corporation and M. Satyanarayanan, School of Computer Science, Carnegie Mellon University</i>	

DEALING WITH THE PC WORLD

Wednesday (2:00 - 3:30)

Chair: Nathaniel S. Borenstein

Wux: UNIX Tools under Windows.....	325
<i>Diomidis Spinellis, Department of Computing, Imperial College of Science, Technology and Medicine</i>	
An MS-DOS Filesystem for UNIX.....	337
<i>Alessandro Forin, Gerald Malan, School of Computer Science, Carnegie Mellon University</i>	
An Overview of the NetWare Operating System.....	355
<i>Greg Minshall, Drew Major, Kyle Powell, Novell, Inc.</i>	

Acknowledgements

Program Committee

Jeffrey C. Mogul, *Program Chair, Digital Equipment Corp. Western Research Laboratory*
Rafael Alonso, *Matsushita Information Technology Laboratory*
Brian N. Bershad, *University of Washington, Department of Computer Science and Engineering*
Nathaniel S. Borenstein, *Bellcore*
Frederick S. Glover, *Digital Equipment Corp. UNIX™ Software Group*
Judith E. Grass, *Corporation for National Research Initiatives*
Michael B. Jones, *Microsoft Research, Microsoft Corporation*
Phil Karn, *Qualcomm, Inc.*
Samuel J. Leffler, *Silicon Graphics, Inc.*
Derek R McAuley, *University of Cambridge*
David Presotto, *AT&T Bell Laboratories*
Margo Seltzer, *Harvard University*
Cathy L. Watkins, *Intel Corp. O/S Technology Engineering*

Scribe

John Chapin, *Department of Computer Science, Stanford University*

Readers

Sean Eric Fagan, *Cygnus Support*
Neil Fishman, *Digital Equipment Corp. Cambridge Research Lab*
J Scott Goldberg, *Rational*
Richard Golding, *Vrije Universiteit, Amsterdam*
Frank D. Greco, *Mercury Technologies, Inc.*
Steven McDowell, *Baker Hughes INTEQ, Inc.*
J. Kent Peacock, *Consultant*
Mary Seabrook, *Fujitsu Open Systems Solutions Inc.*
David B. Wollner, *Xcelerated Systems, Inc.*

Reviewers

Ramón Cáceres
Brad Chen
Bill Cheswick
Steven J. Clark

Simon A Crosby
Fred Douglass
Robin Fairbairns
Jim Hutchison
Brian Marsh

Ken McDonell
Cliff Mercer
Mike O'Dell
John Ousterhout

Preface

People often asked me, in my role as program committee chair, what kinds of papers should appear at a USENIX conference. I think many of us have an implicit idea of the answer, but (unlike, say, for SIGCOMM or for SOSP) it cannot be phrased as covering a specific area of computer science. This is what I told the members of my program committee:

I view the USENIX conference as a sort of "theory meets reality" view of computer systems software. That is, papers should have something new and moderately significant to say, but they should also make sense in the context of real systems, and they should be accessible to an audience of practitioners and experimenters. And, to steal half of the informal IETF motto: "We value working code"; although that's not mandatory, it certainly helps.

The papers we chose for this conference do meet these goals, I think, and I hope you will find them as interesting as we did.

We tried to give each submitted paper thorough consideration. Each paper, in the form of an extended abstract, was reviewed by at least four members of the program committee (or their designees), and at least one of the outside readers. Reviewers scored the papers on several scales, and in almost all cases provided detailed comments. Those papers with ambiguous scores received additional reviews. During the program committee meeting, we discussed every paper individually; we did not accept or reject any paper solely on the basis of its numeric score. The authors of all papers, accepted and rejected, received copies of the reviewer comments, and often some additional comments made during the program committee meeting. About 45% of the accepted papers were "shepherded" by a member of the program committee, to ensure the high quality of the final drafts.

Many people gave a lot of their time and effort, often unpaid, into putting together the refereed program for this conference. Elsewhere in these proceedings you will find a list of the program committee members, readers, and other reviewers who volunteered to read submitted papers. We had a lot of cheerful, efficient support from the USENIX staff, including Carolyn Carr, Cynthia Deno, Judy DesHarnais, Andrea Galleni, Toni Veglia, and Ellie Young; from the Invited Talks coordinators Bob Gray and Brent Welch; and from USENIX old-hands Eric Allman and Rob Kolstad. I thank my employer, Digital Equipment Corporation, and my group, the Western Research Lab, for giving me the time and support to take on the task of program chair. Thanks also to the WRL employees who helped to make our program committee meeting go so smoothly: John Acevedo, Steve Chamberlain, Barbara Hussein, Dennis Miller, Joella Paquette, Catherine Warner, and Annie Warren. And, of course, we would have no program at all if it were not for the people who write and submit papers: thank you all, whether or not we found space for your paper in our program.

Jeffrey Mogul
Western Research Laboratory
Digital Equipment Corporation

FINDING SIMILAR FILES IN A LARGE FILE SYSTEM

Udi Manber¹

Department of Computer Science
University of Arizona
Tucson, AZ 85721
udi@cs.arizona.edu

ABSTRACT

We present a tool, called *sif*, for finding all similar files in a large file system. Files are considered similar if they have significant number of common pieces, even if they are very different otherwise. For example, one file may be contained, possibly with some changes, in another file, or a file may be a reorganization of another file. The running time for finding all groups of similar files, even for as little as 25% similarity, is on the order of 500MB to 1GB an hour. The amount of similarity and several other customized parameters can be determined by the user at a post-processing stage, which is very fast. *Sif* can also be used to very quickly identify all similar files to a query file using a preprocessed index. Application of *sif* can be found in file management, information collecting (to remove duplicates), program reuse, file synchronization, data compression, and maybe even plagiarism detection.

1. Introduction

Our goal is to identify files that came from the same source or contain parts that came from the same source. We say that two files are similar if they contain a significant number of common substrings that are not too small. We would like to find enough common substrings to rule out chance, without requiring too many so that we can detect similarity even if significant parts of the files are different. The two files need not even be similar in size; one file may be contained, possibly with some changes, in the other. The user should be able to indicate the amount of similarity that is sought and also the type of similarity (e.g., files of very different sizes may be ruled out). Similar files may be different versions of the same program, different programs containing a similar procedure, different drafts of an article, etc.

The problem of computing the similarity between two files has been studied extensively and many programs, such as UNIX *diff*, have been developed to solve it. But using *diff* for all pairwise comparisons among, say, 5000 files would require more than 12 million comparisons taking about 5 months of CPU time, assuming 1 second per comparison. Even an order of magnitude improvement in comparison time will still make this

¹ Supported in part by an NSF Presidential Young Investigator Award (grant DCR-8451397), with matching funds from AT&T, by NSF grants CCR-9002351 and CCR-9301129, and by the Advanced Research Projects Agency under contract number DABT63-93-C-0052. Part of this work was done while the author was visiting the University of Washington.

The information contained in this paper does not necessarily reflect the position or the policy of the U.S. Government or other sponsors of this research. No official endorsement should be inferred.

approach much too slow. We present a new approach for this problem based on what we call *approximate fingerprints*. Approximate fingerprints provide a compact representation of a file such that, with high probability, the fingerprints of two similar files are similar (but not necessarily equal), and the fingerprints of two non-similar files are different. *Sif* works in two different modes: all-against-all and one-against-all. The first mode finds all groups of similar files in a large file system and gives a rough indication of the similarity. The running time is essentially linear in the total size of all files² and thus *sif* can be used for large file systems. The second mode compares a given file to a preprocessed *approximate index* of all other files, and determines very quickly (e.g., in 3 seconds for 4000 files of 60MB) all files that are similar to the given file. In both cases, similarity can be detected even if the similar portions constitute as little as 25% of the size of the smaller file.

We foresee several applications for *sif*. The most obvious one is to help in file management. We tested personal file systems and found groups of similar files of many different kinds. The most common ones were different versions of articles and programs (including “temporary” files that became permanent), some of which were generated by the owner of the file system, but some were obtained through the network (in which case, it is much harder to discover that they contain similar information). This information gave us a very interesting view of the file system (e.g., similarity between seemingly unrelated directories) that could not have been obtained otherwise. System administrators can find many uses for *sif*, from saving space to determining whether a version of a given program is already stored somewhere to detecting unauthorized copying. We plan to use *sif* in our work on developing general Internet resource discovery tools [BDMS93]. Identifying similar files (which abound in the Internet FTP space) can improve searching facilities by keeping less to search and giving the users less to browse through. *Sif* can be used as part of a global compression scheme to group similar files together before they are compressed. Yet another important application is in file synchronization for users who keep files on several machines (e.g., work, home, and portable). *Sif* can also be used by professors to detect cheating in homework assignments (although it would be relatively easy to beat it if one wants to put an effort into it), by publishers to detect plagiarism, by politicians to detect many copies of essentially the same form letter they receive from constituents, and so on.

Our notion of similarity throughout this paper is completely syntactic. We make no effort to *understand* the contents of the files. Files containing similar information but using different words will not be considered similar. This approach is therefore very different from the approach taken in the information retrieval literature, and cannot be applied to discover semantic similarities. In a sense, this paper extends the work on approximate string matching (see, for example, our work on *agrep* [WM92a, WM92b]), except that instead of matching strings to large texts, we match parts of large texts to other parts of large texts on a very large scale. Another major difference is that we also solve the all-against-all version of the problem.

A different approach to identifying similarity of source code was taken by Baker [Ba93] who defined two source codes to be similar if one can be obtained from the other by changing parameter names. Baker called this similarity checking *parameterized pattern matching* and presented several algorithms to identify similar source codes. No other differences in the codes were allowed, however. It would be interesting to combine the two approaches.

² A sort, which is not a linear-time routine, is required, but we do not expect it to dominate the running time unless we compare more than 1-2GB of files.

2. Approximate Fingerprints

The idea of computing checksums to detect equal files has been used in many contexts. The addition of duplicate detection to DIALOG was hailed as a “a searcher’s dream come true” [Mi90]. The UNIX *sum* program outputs a 16-bit checksum and the approximate size of a given file. This information is commonly used to ensure that files are received undamaged and untouched. A similar notion of “fingerprinting” a file has been suggested by Rabin [Ra81] as a way to protect the file from unauthorized modifications. The idea is essentially to use a function that maps any size string to a number in a reasonably random way (not unlike hashing), with the use of a secret key. Any change to the file will produce a different fingerprint with high probability. Rabin suggested using 63-bit numbers which lead to extremely low probabilities of false positives. (He also designed a special function that has provable security properties.)

But fingerprints and checksums are good only for *exact* equality testing. Our goal is to identify *similar* files. We want to be able to detect that two files are similar even if their similarity covers as little as 25% of their content. Of course, we would like the user to be able to indicate how much similarity is sought. The basic idea is to use fingerprints on several small parts of the file and have several fingerprints rather than just one. But we cannot use fixed parts of a file (e.g., the middle 10%), because any insertion or deletion from that file will make those parts completely different. We need to be able to “synchronize” the equal parts in two different files and to do that without knowing apriori which files or which parts are involved. We will present techniques that are very effective for natural language texts, source codes, and other types of texts that appear in practice.

We achieve the kind of synchronization described above with the use of what we call *anchors*. An anchor is simply a string of characters, and we will use a fixed set of anchors. The idea is to achieve synchronization by extracting from the text strings that start with anchors. If two files contain an identical piece, and if the piece contains an anchor, then the string around the anchor is identical in the two files. For example, suppose that the string *acte* is an anchor. We search the file for all occurrences of *acte*. We may find the word *character* in which *acte* appears. We then scan the text for a fixed number of characters, say 50, starting from *acte*, and compute a checksum of these 50 characters. We call this checksum a *fingerprint*. The same fingerprint will be generated from all files that contain the same 50 characters, no matter where they are located. Of course, *acte* may not appear in the file at all, or it may appear only in places that have been modified, in which case no common fingerprints will be found. The trick is to use several anchors and to choose them such that they span the files in a reasonably uniform fashion. We devised two different ways to use anchors. The first is by analyzing text from many different files and selecting a fixed set of representative strings, which are quite common but not too common. The string *acte* is an example. Once we have a set of anchors, we scan the files we want to compare and search for all occurrences of all anchors. Fortunately, we can do it reasonably quickly using a our multiple-pattern matching algorithm (which is part of agrep [WM92a]). We will not elaborate too much here on this method of anchor selection, because the second method is much simpler.

The second method computes fingerprints of essentially all possible substrings of a certain length and chooses a subset of these fingerprints based on their values. Again, since two equal substrings will generate the same fingerprints, no matter where they are in the text, we have the synchronization that we need. Note that we cannot simply divide the text into groups of 50 bytes and use their fingerprints, because a single insertion at the beginning of the file will shift everything by 1 and cause all groups, and therefore all fingerprints, to be different. We need to consider *all* 50-byte substrings, including all overlaps. We now present an efficient method to compute all these fingerprints. Denote the text string by $t_1 t_2 \cdots t_n$. The fingerprint for the first 50-byte substring will be

$$F_1 = (t_1 \cdot p^{49} + t_2 \cdot p^{48} + \dots + t_{50}) \bmod M, \text{ where } p \text{ and } M \text{ are constants.}$$

The best way to evaluate a polynomial given its coefficients is by Horner's rule:

$$F_1 = (p \cdot ((\dots (p \cdot (p \cdot t_1 + t_2) + t_3) \dots)) + t_{50}) \bmod M.$$

If we now want to compute F_2 , then we need only to add the last coefficient and remove the first one:

$$F_2 = (p \cdot F_1 + t_{51} - t_1 \cdot p^{49}) \bmod M.$$

We compute a table of all possible values of $(t_i \cdot p^{49}) \bmod M$ for all 256 byte values and use it throughout. Overall, computing all fingerprints is proportional to the number of characters but not to the size of the fingerprint. Deciding which fingerprints to select can be done in many ways, the simplest of them is by taking those with the last k bits equal to 0. Approximately one fingerprint out of 2^k characters will be selected. We use a prime number for p , 2^{30} for M , and $k=8$. Since all selected fingerprints have 8 least significant bits equal to 0, their values should be shifted by 8 before storing them to save space. If the number of files is very large, we may need to use larger fingerprints (i.e., select 2^{31} or 2^{32}) to minimize the number of equal fingerprints by chance.

The second method is easier to use, because the anchors are in a sense universal. They are selected truly at random. It relieves the user from the task of adjusting the anchors to the text. With the first method, anchors that are optimized for Wall Street Journal articles may not be as good for medical articles or computer programs. Anchors for one language may not be good for another language. On the other hand, some users may want to have the ability to fine tune the anchors. For example, with hashing, there is a 1 in 2^k chance (256 in our case) that a string of 50 blanks is selected. If it is, the corresponding fingerprint may appear many times in the file, and it will hardly be representative of the file. The same holds for many other non-representative strings. (We actually encountered that problem; a string of 50 underline symbols turned out to be selected.) One can change the hash function (e.g., by changing p), but there is very little control over the results. One precaution that we take is forbidding overlaps of fingerprints. In other words, once a fingerprint is identified, the text is shifted to the end of it. This way, if, for example, 50 underline symbols form a fingerprint, and the text contains 70 underline symbols, we will not generate 21 duplicate fingerprints.

Both methods are susceptible to bad fingerprints, even for strings that seem representative. The worst example we encountered are fingerprints that are contained in the headers of Postscript files. These headers are large, similar, and ubiquitous; they make many unrelated Postscript files, especially small ones, look very similar. The best solution in this case is to identify the Postscript file when the file is opened and disregard the headers. We discuss handling special files in Section 6.

Although both methods are not perfect, both are good. Having spurious fingerprints is not a major problem as long as there are enough representative fingerprints. Typically, the probability of the same string of 50 bytes appearing in two unrelated files is quite low. And since we require several shared fingerprints we are quite assured of filtering noise. If sufficient number of fingerprints are common to two files then this is a good enough evidence that the two files are similar in some way.

3. Finding Similar Files to a Given File

In this mode, a given file, let's call it the *query file*, is compared to a large set of files that have already been "fingerprinted." The collection of all fingerprints, which we will denote by *All_Fingers*, is maintained in one large file. With each fingerprint we must associate the file it came from. We do that by maintaining the names of all files that were fingerprinted, and associating with each fingerprint the *number* of the corresponding file. The

first thing we do is generate the set *Query_Fingers* of all fingerprints for the query file. We now have to look in *All_Fingers* and compare all fingerprints there to those of *Query_Fingers*. Searching a set is one of the most basic data structure problems and there are many ways to handle it; the most common techniques use hashing or tree structures. In this case, we can also sort both sets and intersect them. But we found that a simple solution using multi-pattern matching was just as effective. We store the fingerprints in *All_Fingers* as we obtained them without providing any other structure, putting one fingerprint together with its file number per line. Then we use *agrep* to search the file *All_Fingers* using *Query_Fingers* as the set of patterns. The output of the search is the list of all common fingerprints to *Query_Fingers* and *All_Fingers*. As long as the set *All_Fingers* is no more than a few megabytes, this search is very effective. (We plan to provide other options for very large indexes.)

Once we have the list of all files containing fingerprints common to the query file, we output those that have more than a given percentage common fingerprints (with default of 50%). This fingerprint percentage number gives a rough estimate for the similarity of the two files. More precisely, it gives an indication of how much of the query file is contained in the file we found. It is interesting to note that this ratio can be greater than 1. That is, the number of common fingerprints to *All_Fingers* and *Query_Fingers* can actually be higher than the total number of fingerprints in *Query_Fingers*. The reason for that is that some fingerprints may appear more than once, and will thus be counted more than once.³

We also provide 32-bit checksums for all files to allow exact comparisons. We compute such checksums together with the fingerprints and store them (along with the file sizes in bytes for extra safety and for more information in the output) with the list of files. We also compute the checksum for the query file and determine whether some files are *exactly* equal to it. This whole process normally takes 2-3 seconds.

4. Comparing All Against All

Comparing all files against all other files is more complicated. It turns out that providing a good way to view the output is one of the major difficulties. The output is a set of sets, or a *hypergraph*, with some similarity relationships among the elements. Hypergraphs are very hard to view (see Harel [Ha88]). We discuss here one approach that is an extension of the one-against-all paradigm, and therefore quite intuitive to view. We experimented with other more complicated approaches and we mention one of them briefly at the end.

The input to the problem is now a directory (or several directories), which we will traverse recursively checking all the files below it, and a threshold number *T* indicating how similar we want the files to be. The first stage is identical to the preprocessing described in the previous section. All files are scanned, all fingerprints are recorded in the fingerprint file *All_Fingers*, and the names, checksums, and sizes of all files are recorded in another file. It will be useful in this stage to separate files according to some types (e.g., text, binary, compressed), using a program such as Essence [HS93], such that files of different types will not be compared. In the current implementation we use only two types, text files and non-text files.

Given *All_Fingers*, we now sort all fingerprints and collate, for each fingerprint associated with more than one file, the list of all file numbers sharing it. The value of the fingerprint itself is not important any more and it is discarded. An example is shown in Figure 1. Removing the fingerprints that appear in no more than one file

³ In fact, this happened to us the very first time we tested *sif*. The query file was an edited version of a call for papers and it matched with 160% "similarity" another file that happened to contain (unintentionally) two slightly different versions of the same call for papers (saved at different times from two mail messages into the same file). Of course, this kind of information is also very useful.

```
10 1174 129 196 647
10 129 196 647
12 213 30 3023 40 4207 44 649 733 942 962
12 213 30 3023 40 4207 44 649 733 942 962
10 129 196 647
11 212 3021 4203 648 732 76 942 961
12 213 30 3023 40 4207 44 649 733 942 962
11 212 3021 4203 648 732 942 961
10 129 196 647
11 212 3021 4203 648 732 942 961
12 213 30 3023 40 4207 44 649 733 77 942 962
11 212 3021 4203 648 732 76 942 961
```

Figure 1: Typical (partial) output of sets of file numbers that share common fingerprints.

reduces the size of *All_Fingers* significantly (for my file system, it was by a factor of 30), so it is much easier to work with. We now have a list of sets and we sort it again, lexicographically, and for sets that appear more than once (meaning the same files share more than one fingerprint) we replace the many copies by one copy and a counter. Figure 2 shows the output of this step, which we denote by *Sorted_Fingers*. Sets with large counters — that is, sets that share significant number of fingerprints — should definitely be part of the output, but how exactly to organize the output turns out to be a very difficult problem.

We are dealing with sets of sets and as a result many complicated scenarios are possible. The similarity relation as we defined it is not transitive. It is possible that file A is similar to file B, which is similar to file C, but A and C have no similarity. You can also have A similar to B, B similar to C, and A similar to C, but the three of

```
1 10 1174 129 196 647
3 10 129 196 647
3 12 213 30 3023 40 4207 44 649 733 942 962
2 11 212 3021 4203 648 732 76 942 961
2 11 212 3021 4203 648 732 942 961
1 12 213 30 3023 40 4207 44 649 733 77 942 962
```

Figure 2: Lines of file numbers that share common fingerprints after sorting and collating. The number in bold is the counter (the number of fingerprints shared by this group of file numbers).

them together, A, B, and C, share no common fingerprints (recall that we allow similarity to correspond to as little as 25% of the files). Or, A, B, C, and D can share 20 fingerprints in common (which is significant), A and C share 40 fingerprints, B and D share 65 fingerprints, A, B, and D share 38 fingerprints, and so on. Does the user really want to see all combinations?

In the first version of *sif* each file *file* that appears somewhere in *Sorted_Fingers* is considered separately. All sets (lines in Figure 2) containing *file* are collected. (This is done while the sets are constructed by associating the corresponding set numbers with each file.) Denote these sets by S_1, S_2, \dots, S_k and their counters by c_1, c_2, \dots, c_k . For each other file that appears in any of the S_i 's, the sum of the corresponding c_i 's is computed. If that sum, as a percentage of the total number of fingerprints that were found for *file*, is more than the threshold, then the file is considered similar to *file*. The output consists of all similar files to *file*, the similarity for each file, and the size of each file. It is easy at this point to skip files whose size differ substantially from that of *file* (if that is what the user wants to see), or files that fall under some other rules specified by the user; for example, only files with the same suffix may be considered similar. One way to reduce the size of this output without losing significant information is to eliminate duplicate sets. If, for example, 7 files are similar, the list of these files will appear 7 times, one for each of them. The similarity percentages may be different in each instance because they are computed as percentages, but the differences are usually minor. So, to reduce the output, any set of files is output no more than once. An example of a partial output is shown in Figure 3.

The following groups of files are similar. Minimum similarity = 25%

```
R100 /u1/udi/xyz/abc/foo.c 10763
    79 /u1/udi/qwe/ewq/bar.c 10979
    75 /u1/udi/uuu/xxx/foobar.c 9560
```

R indicates the file with which all others are compared. The first number is the percentage of similarity with the R file. The numbers at the end indicate the file sizes. (Except for the file names, this is an actual output.)

Figure 3: An example of a group of similar files as output by *sif*.

The second version of *sif* will include as an option a list of *interesting* similar files, where a set is considered interesting if all members of the set have sufficient number of common fingerprints *and* the set is not a subset of a larger interesting set. The problem of generating interesting sets turns out to be NP-complete, but we devised an algorithm that we hope will work reasonably well in practice.

5. Experience and Performance

As expected, *sif* performs very well on random tests. For example, we took a file containing a C program of size 30K, and made 300 random substitutions (with repetition), each of size 50, thus changing about 50% of the file (but leaving significant chunks unchanged). We then ran *sif* (in the one-against-all mode) setting the threshold at a very low 5%. We ran this experiment 50 times. Each time *sif* found the right file (among 4000 other files of about 60MB) and only it. The similarity that *sif* reported ranged from 37% to 62%, averaging 52%. The average

running time for one test (user + system time) was 3.1 seconds (not counting, of course, the time it took originally to build the index). (All experiments were run on a DEC 5000/240 workstation running Ultrix.)

The real question, however, is the performance on real data. We tried *sif* on several file systems. The running time for computing all fingerprints was 3-6 seconds per MB which is in the order of 500MB to 1GB per hour. It takes longer if there are many files and directories and they are small. The sorting of the fingerprints takes from a third to a half of this time (sorting is not a linear-time algorithm, so it takes longer for large number of fingerprints; we tried up to 200MB which generated 800,000 fingerprints). The rest of the algorithm takes much less time.

The first run was on a file system with 2750 (uncompressed) text files of about 40MB. It took 127.4 seconds (user + system time) to generate all fingerprints (and determine that 800 other files are non-text files), 74.3 seconds to sort the fingerprints, and 14.6 seconds to perform the rest of the computation (only 1 second of which depends on the similarity parameter, so changing it will take just one more second literally). Another test was performed on a large collection of "Frequently Asked Questions" (FAQ) files extracted automatically from many newsgroups. For example, two FAQs that were archived under different names (misc.quotes and misc.quotations.sources) turned out to be almost the same.

The next collection of files presented us with almost the worst case. We obtained, through the Alex system [Ca92], a large collection of 21249 README files taken from thousands of ftp sites across the Internet. Their total size was 73MB, which puts the average size at a very low 3K. We found 3620 groups of equal files and 2810 other groups of similar files (the similarity threshold was set at 50%). In many cases it would be impossible to tell that the files are similar by looking only at their names (e.g., draco.ccs.yorku.ca:pub/doc/tmp/read.me and ee.utah.edu:rhc/README are very similar but not the same). The most challenging experiment was the whole X11R5 distribution (380MB, ~200MB of which were 21288 ascii files). There were 657 groups of equal files, 3445 groups of similar files with 25% similarity threshold, and 2915 groups with 50% threshold. For most of the experiments described here (with the exception of the FAQ files), we used the first method with a list of very frequent anchors to allow high precision. On the average, one fingerprint was generated for about every 200 bytes of text.

The space required to hold the fingerprints for the one-against-all tool is currently about 5% of the total space. By a better encoding of the fingerprints this figure will be reduced to about 2%.

6. Future Work

We are extending *sif* in four areas. The first is adapting the fingerprint generation to file types. We already mentioned Postscript files, for which the headers should be excluded from generating fingerprints. This is relatively easy to do because Postscript files and their headers can be easily recognized. Another example is removing formatting statements from files (e.g., troff or TeX files). Other types present more difficult problems. The two most notable types we would like to handle are executables and compressed files. Both types are very sensitive to change. Adding one statement to a program can change all addresses in the executable. In compressed files that translate strings to dictionary indices (e.g., Lempel-Ziv compression) one change in the dictionary can change all indices. The challenge is to find invariants and generate fingerprints accordingly (e.g., ignoring addresses altogether, exploring the relationships between dictionary indices).

The second area is allowing different treatments of small and large files. Currently, we treat all files equally. A noble idea, but sometimes not effective. We figured that at least 5-10 shared fingerprints are needed

for a strong evidence of similarity (the exact number depends on the file type). If we seek 50% similarity, then each file needs at least 10-20 fingerprints. In the current setting (which can be easily changed), *sif* generates 3-4 fingerprints per 1K, which makes *sif* only marginally effective for files of less than 5K. On the other hand, a file of 1MB can generate 4,000 fingerprints. If we adjust the number of fingerprints to the size of the file, we may lose the ability to determine whether a small file is contained in a large file, but this is not always needed. Adjusting the number of fingerprints is easy with both methods of anchor selection (by decreasing the number of anchors with the first method or increasing the value of k with the second method).

The third area is providing convenient facilities for comparing two directories. Current tools for comparing directories (such as the system V *dircpm* program) rely on file names and checksums. For files that do not match exactly, *dircpm* will only list them as not being equal. Comparing two directories based on content can be done with essentially the same tools we already have, but we need to allow the use of the filenames in the similarity measure.

The fourth area is customizing the output generation. The most difficult problem is how to provide users with flexible means to extract only the similarity they seek. We could attach a large number of options to *sif* (a popular UNIX solution) and provide hooks to external filter routines (such as ones using Essence [HS93]). We would like to have something more general. The problem of finding *interesting* similar files (as defined at the end of Section 4) is very intriguing from a practical and also a theoretical point of view.

Acknowledgements

We thank Vincent Cate for supplying us with the README files collected by Alex. Richard Schroepel suggested the use of hashing to select anchors. Gregg Townsend wrote the filter that collects the FAQ files.

References

[Ba93]

Baker, B. S., "A theory of parameterized pattern matching: Algorithms and applications," *25th Annual ACM Symposium on Theory of Computing*, San Diego, CA (May 1993), pp. 71-80.

[BDMS93]

Bowman, C. M., P. B. Danzig, U. Manber, and M. F. Schwartz, "Scalable Internet Resource Discovery: Research Problems and Approaches," University of Colorado Technical Report# CU-CS-679-93 (October 1993), submitted for publication.

[Ca92]

Cate, V., "Alex — a global filesystem," *Proceedings of the Usenix File Systems Workshop*, pp. 1-11, May 1992.

[Ha88]

Harel D., "On Visual Formalisms," *Communications of the ACM*, 31 (May 1988), pp. 514-530.

[HS93]

Hardy D. R., and M. F. Schwartz, "Essence: A resource discovery system based on semantic file indexing," *USENIX Winter 1993 Technical Conference*, San Diego (January 1993), pp. 361-374.

[Mi90]

Miller C., "Detecting duplicates: a searcher's dream come true" *Online*, 14, 4 (July 1990), pp. 27-34.

[Ra81]

Rabin, M. O., "Fingerprinting by Random Polynomials," Center for Research in Computing Technology, Harvard University, Report TR-15-81, 1981.

[WM92a]

Wu S. and U. Manber, "Agrep — A Fast Approximate Pattern-Matching Tool," *Usenix Winter 1992 Technical Conference*, San Francisco (January 1992), pp. 153–162.

[WM92b]

Wu S., and U. Manber, "Fast Text Searching Allowing Errors," *Communications of the ACM* 35 (October 1992), pp. 83–91.

Biographical Sketch

Udi Manber is a professor of computer science at the University of Arizona, where he has been since 1987. He received his Ph.D. degree in computer science from the University of Washington in 1982. His research interests include design of algorithms, pattern matching, computer networks, and software tools. He is the author of "Introduction to Algorithms - A Creative Approach" (Addison-Wesley, 1989), and the editor of 3 other books. He received a Presidential Young Investigator Award in 1985.

cql – A Flat File Database Query Language

Glenn Fowler
gsf@research.att.com

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Abstract

cql is a UNIX* system tool that applies C style query expressions to flat file databases. In some respects it is yet another addition to the toolbox of programmable file filters: *grep* [Hume88], *sh* [Bour78][BK89], *awk* [AKW88], and *perl* [Wall]. However, by restricting its problem domain, *cql* takes advantage of optimizations not available to these more general purpose tools.

This paper describes the *cql* data description and query language, query optimizations, and provides comparisons with other tools.

1. Introduction

Flat file databases are common in UNIX system environments. They consist of newline terminated records with a single character that delimits fields within each record. Well known examples are */etc/passwd* and */etc/group*, and more recently the *sablime* [CF88] MR databases and *cia* [Chen89] abstraction databases.

There are two basic flat file database operations:

update – delete, add or modify records

query – scan for records based on field selection function

For the most part UNIX system tools make a clear distinction between these operations. Update is usually done by special purpose tools to avoid problems that arise from concurrency. Some of these tools are admittedly low-tech: *vipw* write locks the */etc/passwd* file and runs the *vi* editor on it; any other user running *vipw* concurrently will be locked out. On the other hand query tools usually assume that the input files are readonly or that they at least will not change during query access. *cql* falls into this category: it is strictly for queries and supports no update operations. Despite this restriction *cql* adequately fills the gap between *awk* and full featured database management systems.

In the simplest case a flat file database query is a pattern match that is applied to one or more fields in each record. The output is normally a list of all matched records. *grep*, *sh*, *awk*, and *perl* are well suited for such queries on small databases. These commands scan the database from the top, one record at a time, and apply the match expression to each record. Unfortunately, as the number of records and queries increases, the repeated linear scans required by these tools soon become an intolerable bottleneck. The bottleneck can be diminished by examining the queries to limit the number of records that must be scanned, but this requires some modifications, either to the database or to the scanning tools.

Some applications, such as *sablime*, ease the bottleneck by partitioning the database into several flat files based on one or more of the record fields. This speeds up queries that key on the partitioned fields, but hinders queries that must span the partition. Besides complicating the application query implementation, partitioning also

* UNIX is a registered trademark of USL.

imposes complexity on database updates and backup.

The *perl* solution (actually, *one* of the *perl* solutions – *perl* is the UNIX system swiss army knife) is to base the queries on *dbm* [BSD86] hashed files rather than flat files. Linear scans are then avoided by accessing the *dbm* files as associative arrays. A problem with this is that a *dbm* file contains the hashed field name and record data for each database record, so its file size is always larger than the original flat file. This method also generates a separate *dbm* file for each hashed field, making it unacceptable for use with large databases.

Other applications, such as *cia*, preprocess the database by generating B-tree or hash index files [Park91] for quick random access. Specialized scanning tools are then used to process the queries. The advantage here is that no database changes are required to speed up the queries. In addition, hash index files only store pointers into the database, so their size is usually smaller than the original database. The speedup, though, is not without cost. Some of the tools may be so specialized as to work for only a small class of possible queries; new query classes may require new tools.

Along with sufficient access speed, another challenge is to provide reasonable syntax and semantics for query expressions. For maximal transparency and portability the database fields should be accessed by name rather than number or position. Otherwise queries would become outdated as the database changes.

cql addresses these issues by providing a fast, interpreted symbolic interface at the user level, with automatic record hash indexing and query optimization at the implementation level. Query expressions are modeled on C, including a **struct** construct for defining database record schemas.

2. Background

As opposed to the UNIX system database tools like *unity* [Felt82], *cql* traces its roots to the C language and the *grep* and *awk* tools. As such *cql* is limited to readonly database access.

An example will clarify the differences between the various tools. The example database is */etc/passwd* with the record schema:

```
name:passwd:uid:gid:info:home:shell
```

where `:` is the field delimiter, *uid* and *gid* are numeric fields, and the remaining fields are strings. The example query selects all records with *uid* less than 10 and no *passwd*.

Example solutions may not be optimal for each tool, but they are a fair representation of what can be derived from the manuals and documentation. The author has a few years experience with *grep* and *sh*, some exposure to *awk*, but had to resort to a netnews request for *perl*.

2.1 *grep*

```
grep '^([^\:]*):[0-9]:' /etc/passwd
```

grep associates records with lines and has no implicit field support, so the select expression must explicitly list all fields. As it turns out the expression *uid*<10 can be matched by a regular expression; more complicated expressions would require extra tool plumbing, possibly using the *cut* and *expr* commands. *grep* differs from the other tools in that a single regular expression pattern describes both the schema and query. This works fine at the implementation level but is cumbersome as a general purpose user interface.

2.2 *awk*

```
awk '
BEGIN { FS = ":" }
{ if ($3 < 10 && $2 == "") print }
' /etc/passwd
```

Lines are the default *awk* record and *FS* specifies the field separator character. Numeric expressions are as in C and string comparison may also use the `==` and `!=` operators. Unfortunately the fields are named by number (starting at 1). If the database format changes then all references to *\$number* must be changed accordingly. An advantage over *grep* is that fields are accessed as separate entities rather than being a part of the matching

pattern.

2.3 shell

```
ifs=$IFS
IFS=:
while read name passwd uid gid info home shell junk
do   if (( $uid < 10 )) && [[ $passwd == "" ]]
    then IFS=$ifs
        print "$name:$passwd:$uid:$gid:$info:$home:$shell"
        IFS=:
    fi
done < /etc/passwd
```

The shell (*ksh*[BK89]) version uses the field splitting effects of *IFS* and *read* to blast the input records. A nice side effect is that *read* also names the fields. If the database changes then only the field name arguments to *read* must change. Notice, however, that the shell has different syntax for numeric and string comparisons. Also, older shells [Bour78] have no built-in expressions and would require a separate program like *expr* to do the record selection.

2.4 perl

```
perl -e '
    open (PASSWD, "< /etc/passwd") || die "cannot open /etc/passwd: $!";
    while (<PASSWD>) {
        ($name, $passwd, $uid, $gid, $info, $home, $shell) = split(":");
        if ($uid < 10 && $passwd eq "") {
            print "$name:$passwd:$uid:$info:$home:$shell";
        }
    }
'
```

The *perl* example [Chri92] is similar to *shell*, except that *shell* combines the record read and field split operations into a single *read* operation. As with *shell* string equality requires special syntax and *\$* must prefix expression identifiers.

2.5 cql

```
cql -d "
    passwd {
        char*    name;
        char*    passwd;
        int      uid, gid;
        char*    info;
        char*    home, shell;
    }
    passwd.delimiter = ':';
" -e "uid < 10 && passwd == ''" /etc/passwd
```

cql queries are split into two parts. The *declaration* section (-d) describes the record schema and the *expression* section (-e) provides the matching query. Using *cql* for this query is overkill, but it provides a basis for the more complex examples that follow.

2.6 Performance

Figure 1 shows the timing in user+sys seconds for the above examples, ordered from best to worst. The times were averaged over 5 runs on a lightly loaded 20 mip workstation with 2 cpus on an input file consisting of 19,847 records (1,525,549 bytes) in a local file system. *cat* is included as a lower bound.

<i>cat</i>	0.31
<i>grep</i>	1.77
<i>cql</i>	3.29
<i>awkcc</i>	3.37
<i>awk</i>	7.73
<i>perl</i>	9.09
<i>ksh</i>	19.98

Figure 1. Example timings

Although the compiled *awkcc* example runs more than twice as fast as the *awk* script it suffers by having a fixed select expression. Any change in the expression would force recompilation of the *awk* script to make a new executable. The timings also show that performance for the example query seems to be inversely proportional to tool functionality.

3. Optimization

Queries that check fields for equality are candidates for optimization. For example, most */etc/passwd* queries are lookups for a particular name, uid or gid. As mentioned before, *perl* supports an associative array interface to *dbm* hash files, but converting to use this would require more than four times the file space of */etc/passwd* itself and the query syntax would need to change to use the array notation. *cql* offers an alternative that only changes the schema declaration:

```
passwd {
    register char*  name;
    char*          passwd;
    register int    uid, gid;
    char*          info;
    char*          home, shell;
}
```

As with C the **register** keyword is a hint that marks variables that may be frequently accessed. For *cql* **register** marks fields that may be frequently checked for equality. *cql* generates a hash index file for each **register** field during the first database query. Subsequent queries use the index files to prune the scan to only those records with the same hash value as the **register** fields in the query expression. The index files are connected to a particular database; if the database file changes then the index files are regenerated by doing a full database scan. Because of index file generation the first query on schemas with **register** fields is always slower than subsequent queries.

The hash index file algorithm is due to David Korn and has been implemented as a library (*hix*) by the author. A *hix* file stores only hash codes and database file offsets, and its size ranges from 10% to 50% of the original database. The */etc/passwd* example above has one record with the name **bozo**. The timings for the query `name=="bozo"` are listed in Figure 2.

no register fields	2.95
first register query	6.52
subsequent register queries	0.54
grep	1.64
awk	7.13
perl	7.56

Figure 2. Register query timings

The *hix* file generation slowed the first query by over 2 times but the subsequent queries were about 10 times faster. Even with *hix* file generation *cql* is still slightly faster than *awk* and *perl*. For the example */etc/passwd* file size of 1,525,644 bytes the 3 *hix* files were a total of 788,952 bytes, or approximately 50% of the original database size.

4. Sub-schemas

Fields often contain data that can be viewed as another database record. *cql* supports this by allowing schema fields within schemas. The sub-schema fields are then accessed using the familiar C `'.'` notation. Our local */etc/passwd* file formats the info field as:

```
info {
    char*  name, address, office, home;
}
info.delimiter = ",";
```

where the info sub-schema delimiter is `','`. An important difference with C declaration syntax is that the *cql* `char*` is a basic type. This means that all of the fields in this example have type `char*`, whereas in C only the first field would be `char*`.

Adding a second schema declaration introduces an ambiguity as to which schema applies to the main database file. By default the main schema is first schema from the top. `schema=schema-name;` can be used to override the default. The complete declaration now becomes:

```
passwd {
    register char*  name;
    char*          passwd;
    register int    uid, gid;
    info           info;
    char*          home, shell;
}
info {
    char*  name, address, office, home;
}
passwd.delimiter = ":";
info.delimiter = ",";
schema = passwd;
```

and the following queries are possible:

```
info.name=="Bozo T. Clown"
info.address=="* MH *"
```

where the second query illustrates *ksh* pattern matching on the address field.

Fields that refer to sub-schema data in different files are also possible. In this case the sub-schema field data is actually a key that corresponds to a field (usually the first) in the sub-schema data file. *cia* uses this format for its **reference** and **symbol** schemas. Sub-schema pointers are also used in *shadow password* implementations that split the */etc/passwd* file into */etc/passwd* that contains public information (no encrypted passwords) and

/etc/shadow that contains privileged information (the encrypted passwords). An example **/etc/passwd** record might look like:

```
bozo:*shadow*:123:123:Bozo T. Clown, Big Top, 123-456::
```

with a corresponding **/etc/shadow** record:

```
bozo:abcdef.FEDCBA:Aug 11, 1993
```

In this case the name field doubles as the shadow key and the main schema **passwd** field is ignored. The *cql* declaration for these shadow passwords is:

```
passwd {
    shadow*      name;
    char*        passwd;
    register int uid, gid;
    info         info;
    char*        home, shell;
}
info {
    char* name, address, office, home;
}
shadow {
    char*      name;
    char*      passwd;
    date_t     expire;
}
delimiter = ":";
info.delimiter = ",";
schema = passwd;
passwd.input = "/etc/passwd";
shadow.input = "/etc/shadow";
```

C pointer notation is used to declare the shadow sub-schema reference and the predefined `date_t` type (described below) is used for the shadow password expiration field. The sub-schema reference also requires an input file that can be assigned by a *schema="pathname"*; statement. A shadow equivalent of the original example query is:

```
uid<10 && name.passwd==" "
```

Notice that ``.`` is also used to dereference sub-schema pointers (as opposed to ``->'` in C). Additionally, **register** is inferred for all sub-schema pointers. This allows *cql* to optimize queries by transforming equality expressions on sub-schema fields into hash index offset expressions. The **extern** keyword denotes the sub-schema key field in the sub-schema declaration; it may be omitted if the key field is the first sub-schema field.

Sub-schema data can also be specified in the declaration section:

```

info {
    char*    name;
    map*     type;
}
map {
    char*    name;
    char*    value;
}
info.type.input = { /* each "string" is a record */
    "_ERROR_",    "g;global",    "t;typedef",
    "e;extern",   "s;static",    "l;libsym"
};

```

This is convenient for expanding database encodings in the user interface. For example, `type.value=="extern"` is more descriptive than `type=="e"`.

5. Language Description

The declaration language has already been introduced in the previous examples. Schemas are declared using the C **struct** style:

```

schema-name
{
    type-specifier    field-name [ , field-name ... ] ;
    ...
}

```

Schema, type and field names must match the regular expression `[a-zA-Z_][a-zA-Z_0-9]*`. The C reserved words `break`, `case`, `continue`, `default`, `else`, `for`, `if`, `return`, `switch`, `while`, are also reserved in *cql*, as are the following predefined types:

<code>char*</code>	A variable length string.
<code>date_t</code>	A date represented internally as seconds since the epoch. The data representation can be in any of the "standard" forms. <code>date_t</code> fields can be used in date comparisons such as <code>mtime<"yesterday"</code> .
<code>double</code>	A double floating point constant.
<code>elapsed_t</code>	Scaled elapsed time showing the two most significant time components. Examples are: 1.03s one and three one hundredth seconds 2m20s two minutes and 20 seconds 3w11d three weeks and 11 days
<code>float</code>	Equivalent to <code>double</code> .
<code>int</code>	Equivalent to <code>long</code> .
<code>long</code>	A long integer constant.
<code>void</code>	The return type of user defined actions.

A *type-specifier* may be schema name. *schema-name** declares a sub-schema field whose data is in a separate file whereas *schema-name* declares a sub-schema whose data is the field data itself. *type-specifier field-name[size]* declares an array field whose values are separated by a sub-field delimiter.

The schema name *cql* is also reserved. The fields in this schema are predefined by *cql* and provide access to run-time information. The fields are:

`elapsed_t clock` The elapsed time since the start of this *cql* in hundredths of a second.

`date_t date` The time this *cql* started.

`int errors` The non-fatal error count.

`char* getenv(char* name)` Returns the value of the environment variable *name*. For example, `cql.getenv("PATH")` is the value of the `PATH` environment variable.

`int line` The current declaration or expression input line.

`int offset` The current record offset in the main schema input file starting at 0.

`char* path(char* name, int length)` Returns value of the file pathname *name* truncated to *length*. The truncation attempts to preserve the significant parts of the pathname.

`int record` The current record number in the main schema input file starting at 1.

`int select` The number of records selected by the select expression.

`int size` The size in bytes of the current input record.

`date_t time` The current time.

Schema attributes are also assigned in the declaration section.

```
schema = schema-name;
```

sets the main schema name. If omitted the main schema defaults to the first declared schema (from the top). Schema delimiters are defined by:

```
qualified-schema-name.delimiter = "delimiter";
```

The default main schema delimiter is `:` and the default sub-schema delimiter is `;`. Schema input is defined by:

```
qualified-schema-name.delimiter = "path";
```

where *path* is the pathname of the schema data file, or by:

```
qualified-schema-name.input = {
    "record-1",
    "record-n",
};
```

where *record-i* are the schema records. Input data may be shared by:

```
a.input = b.input = ...
```

The default main schema input is the standard input; indirect sub-schema inputs must always be defined.

The expression language is basically C with the exception that `char*` operands are allowed for the `==`, `!=`, `<`, `<=`, `>=`, `>` operators. The expression may be labeled by:

```
label: expression;
...
```

or equivalently by:

```
void label() { expression };
...
```

the former being more convenient for command line expressions. The value of a labelled expression is either the value of a `return` statement evaluated in the expression or the value of the last statement evaluated in the expression. The default expression label is `select`. The `select` expression is applied to each record to determine the matching records. The default `select` is 1, i.e., all records match. The `action` expression is then applied to each record matched by `select`. The default `action` lists the matching records on the standard output. The `begin` expression, if specified, is evaluated before the database is scanned and the `end` expression, if specified, is

evaluated after all records have been scanned.

It is important to note that *cql* is not a complete C interpreter. Just enough is borrowed from C to get the query job done. The implementation is based on a C expression library that was originally written for the *rw* [FKV89] replacement for the *find* command. This library constitutes a large portion of the work; in fact the *cql* prototype was written in one day (although the addition of hash indexing and query optimization took considerably longer).

6. Reports

A query language is incomplete without some form of reporting. *cql* provides the `printf` function to output field values. `printf` is most often used in **action** expressions to output portions of selected fields. For example,

```
select: uid < 10 && name.expire < "in 2 days";
action: printf("%s will expire on %s\n", name, name.expire);
```

Format specifications are type checked and string to integer conversions are supported. For example, `name.expire` printed as `%d` lists the seconds since the epoch, but printed as `%s` lists the date in the standard date format.

Headers and footers are done by using `printf` in the **begin** and **end** expressions. Fancier operations like pagination, text filling and font highlights are left to tools designed specifically for that job.

Selected records may be sorted by specifying the sort order:

```
sort = { field, ... };
```

7. Alternate Database Formats

Three alternate database formats are supported. The formats are orthogonal and may appear in any combination. The first, *virtual database files*, **Figure 3**, allows multiple schemas to be defined in a single file. This format is similar to UNIX system archive file format but is designed for quick access to the individual schemas. In the past *cia* generated two database files with the fixed names **reference.db** and **symbol.db**. This made it difficult to copy and backup abstraction databases. By using a virtual database *cia* now combines the two files into a single file with an application specific name, e.g., **ksh.db** for the *ksh* abstraction database. A virtual database file acts like a directory in the *cql* interface. For example, the `symbol` schema data in the `ksh.db` virtual file is named `ksh.db/symbol`. A virtual database file is also used to store the index files for each input database. The index file is named by inverting the case of the corresponding database file name suffix. For example, the indices for `ksh.db` would be `ksh.DB`.

The second format, *partitioned database files*, **Figure 4**, allows a single database schema to span more than one file. The implementation is complicated by the fact that separate index files may be associated with each partition. Partitions are named as a single file pathname by separating the partition component pathnames with `':'`, as in `"ksh.db:libast.db:libdl.db"`.

Finally, a *union database file*, **Figure 4**, allows schema records to be intermixed in a single file. The first union record field is used to identify the schema for each record. This format is convenient for applications that generate streams of different record types.

8. Future Work

It should be possible to access binary and fixed length fields with no delimiters. These are not supported yet.

A transitive closure operation would also be useful.

9. Conclusion

cql is an attractive scripting tool for database queries. Because it is an interpreted query language it supports rapid prototyping; because it is fast it allows prototypes to become production code. Its interpreted nature also makes it easy for database users (as opposed to database providers) to write and test new queries.

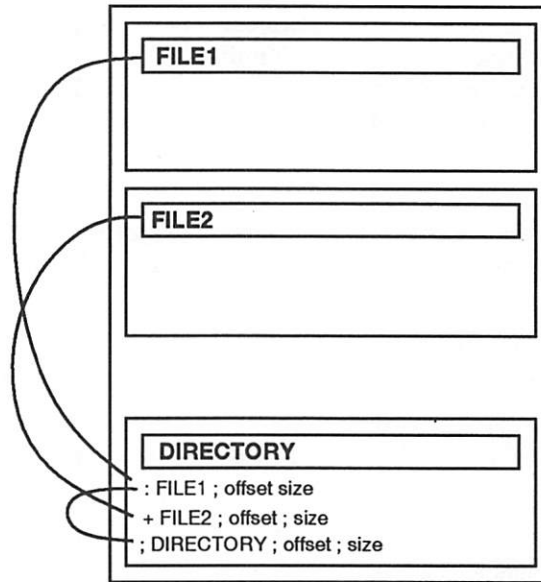
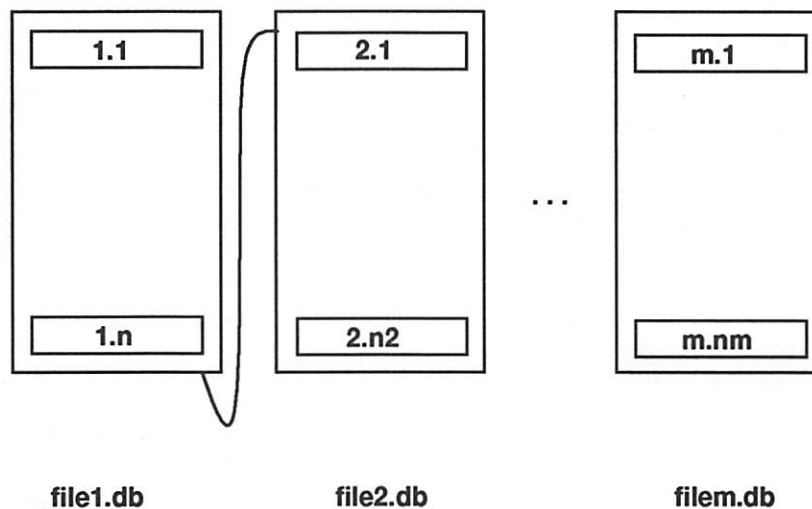


Figure 3. Virtual database format



file1.db:file2.db: . . . :filem.db

Figure 4. Partition database format

10. Acknowledgements

Thanks go to my colleagues in the Software Engineering Research Department: Phong Vo for the excellent *sfl* implementation that pepped up *cql* I/O; David Korn for the hash index file algorithm; Robin Chen for translating database lingo and concepts into terms a shell and C hack could understand.

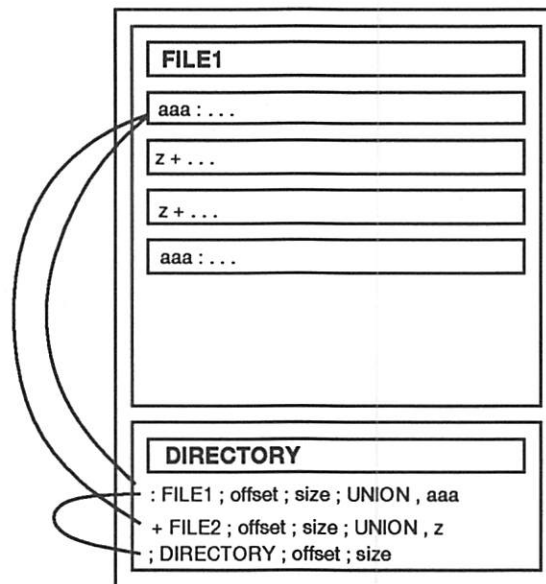


Figure 5. Union database format

References

- [AKW88] A. V. Aho, B. W. Kernighan, P. J. Weinberger, *The Awk Programming Language*, Addison-Wesley, 1988.
- [BK89] Morris Bolsky and David G. Korn, *The KornShell Command and Programming Language*, Prentice Hall, 1989.
- [Bour78] S. R. Bourne, *The UNIX Shell*, AT&T Bell Laboratories Technical Journal, Vol. 57 No. 6 Part 2, pp. 1971-1990, July-August 1978.
- [BSD86] *The UNIX Programmer's Reference Manual: 4.3 Berkeley Software Distribution*, UC Berkeley, California, 1986.
- [CF88] Steve Cichinski and Glenn S. Fowler, *Product Administration through Sable and Nmake*, AT&T Bell Laboratories Technical Journal, Vol. 67 No. 4, pp. 59-70, July-August 1988.
- [Chen89] Yih-Farn Chen, *The C Program Database and Its Applications*, Proc. of Summer 1989 USENIX Conf.
- [Chri92] Tom Christiansen, private correspondence.
- [Felt82] S. Felts, *The Unity DBMS*, AT&T Bell Laboratories Technical Memorandum, TM82-59312-1, 1992.
- [FKV89] Glenn S. Fowler, David G. Korn and Kiem-Phong Vo, *An Efficient File Hierarchy Walker*, Proc. of Summer 1989 USENIX Conf.
- [Hume88] Andrew Hume, *Grep Wars*, EUUG Conference Proceedings, London, England, Spring 1988.
- [Park91] *Off The shelf: B-tree data-file managers*, Tim Parker, UNIX Review, Vol. 9 No. 3, pp. 55-58, March 1991.
- [Wall] Larry Wall, *The Nutshell perl Book*.

Glenn Fowler is a distinguished member of technical staff in the Software Engineering Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey. He is currently involved with research on configuration management and software portability, and is the author of *nmake*, a configurable ANSI C preprocessor library, and the *coshell* network execution service. Glenn has been with Bell Labs since 1979 and has a B.S.E.E., M.S.E.E., and a Ph.D. in electrical engineering, all from Virginia Tech, Blacksburg Virginia.

GLIMPSE: A Tool to Search Through Entire File Systems

Udi Manber¹

Department of Computer Science
University of Arizona
Tucson, AZ 85721
udi@cs.arizona.edu

Sun Wu

Department of Computer Science
National Chung-Cheng University
Ming-Shong, Chia-Yi, Taiwan
sw@cs.ccu.edu.tw

ABSTRACT

GLIMPSE, which stands for GLoBal IMPLICIT SEarch, provides indexing and query schemes for file systems. The novelty of *glimpse* is that it uses a very small index — in most cases 2-4% of the size of the text — and still allows very flexible full-text retrieval including Boolean queries, approximate matching (i.e., allowing misspelling), and even searching for regular expressions. In a sense, *glimpse* extends *agrep* to entire file systems, while preserving most of its functionality and simplicity. Query times are typically slower than with inverted indexes, but they are still fast enough for many applications. For example, it took 5 seconds of CPU time to find all 19 occurrences of *Usenix AND Winter* in a file system containing 69MB of text spanning 4300 files. *Glimpse* is particularly designed for personal information, such as one's own file system. The main characteristic of personal information is that it is non-uniform and includes many types of documents. An information retrieval system for personal information should support many types of queries, flexible interaction, low overhead, and customization. All these are important features of *glimpse*.

¹ Supported in part by an NSF Presidential Young Investigator Award (grant DCR-8451397), with matching funds from AT&T, by NSF grants CCR-9002351 and CCR-9301129, and by the Advanced Research Projects Agency under contract number DABT63-93-C-0052. Part of this work was done while the author was visiting the University of Washington.

The information contained in this paper does not necessarily reflect the position or the policy of the U.S. Government or other sponsors of this research. No official endorsement should be inferred.

1. Introduction

With the explosion of available information, especially through networks, organizing even one's own personal file system is becoming difficult. It is hard, even with a good organization, to remember things from a few years back. (Now where did I put the notes on that interesting colloquium two years ago?) After a while, not only the content of a piece of information can be forgotten, but also the *existence* of that information. There are two types of tools to search for patterns: grep-like tools, which are fast only if the search is limited to a small area, and index-based tools, which typically require a large index that needs to be computed ahead of time. *Glimpse* is a combination of the two approaches. It is index-based, but it uses a very small index; it assumes no structure from the data; and it allows very flexible queries including approximate matching.

The most common data structure used in information retrieval (IR) systems is an inverted index [SM83]. All occurrences of each word are stored in a table indexed by that word using a hash table or a tree structure. To reduce the size of the table, common words (e.g., *the* or *that* in English) are sometimes not indexed (although this cannot be done when the text is multi-lingual). Inverted indexes allow very fast queries: There is no need to search in any of the texts, only the table needs to be consulted and the places where the word occurs are retrieved immediately. Boolean queries are slower, but are still relatively fast.

The main drawback of inverted indexes for personal file systems is their space requirement. The size of the index is typically 50%-300% of the size of the text [Fa85]. This may not be a major drawback for commercial text databases, because disk space is relatively cheap, but it is a major drawback for personal information. Most users would not agree to double their disk cost for the benefit of indexing. Indeed today most personal file systems are not indexed. But, due to an increased availability of digital information through networks, many personal file systems are large enough to require IR capabilities. Signatures files [Fa85, GB91] have been suggested as an alternative to inverted indexes with good results. Their indexes are only 10%-30% of the text size. Their search time is slower than with inverted indexes, and since they are based on hashing, their parameters must be chosen carefully (especially for many files of different sizes) to minimize the false drops probability.

The second weakness of inverted indexes is the need for exact spelling (due to the use of hashing or tree structures to search for keywords in the index efficiently). If one is looking for all articles containing Schwarzkopf for example, any article with a misspelling will be missed (not to mention that the exact spelling is needed to form the query). The typical way to find misspelled words is to try different possibilities by hand, which is frustrating, time consuming, and is not guaranteed to succeed. In some cases, especially in large information bases, searching provides the only way to access information. A misspelling can cause a piece of information to be virtually lost. This is the digital equivalence of dropping a folder behind the file cabinet (except that there is no limit to the room behind this 'file cabinet,' and hardly anyone ever cleans there). This problem is expected to become more acute as more information is scanned by OCR (Optical Character Recognition) devices, which currently have an error rate of 1-5% [BN91, RKN92].

Glimpse requires a very small index, in most cases 2-4% of the original text, and it supports arbitrary approximate matching. As we will describe in detail in the next section, *glimpse*'s engine is *agrep*, a search program that we developed [WM92a, WM92b], which allows the user to specify the number of allowable errors (insertions, deletions, substitutions, or any combination). The user can also search for the best match, which will automatically find all matches with the minimum number of errors. Because we use a small index, our algorithms are usually slower than ones using inverted indexes, but their speed is still on the order of a few seconds, which is fast enough for single users.

In some sense, *glimpse* takes the opposite extreme to inverted files in the time vs. space tradeoff (with signature files being in the middle). For some applications, such as management of personal information, speed is a secondary issue. Most users would rather wait for 10-15 seconds, or even longer, for a query than double their disk space. Even for IR systems, such as library card catalogs, where high throughput is essential, our scheme can be used as a secondary mechanism to catch spelling errors. For example, we found several spelling errors in a library catalog (see Section 4). We believe that this capability is essential in all applications that require a high level of reliability; for example, medical labs could miss information on patients due to misspelling.

We call our method *two-level searching*. The idea is a hybrid between full inverted indexes and sequential search with no indexing. It is based on the observation that with current computing performance, sequential search is fast enough for text of size up to several megabytes. Therefore, there is no need to index every word with an exact location. In the two-level scheme the index does not provide exact locations, but only pointers to an area where the answer may be found. Then, a flexible sequential search is used to find the exact answer and present it to the user. (Some other IR systems, such as MEDLARS [SM83] and STATUS [Te82], allow sequential search as a postprocessing step to further filter the output of a query, but the search relies on inverted indexes.) The idea of two-level searching is quite natural, and, although we have not found references to it, it has most probably been used before. The use of *agrep* for both levels — searching the index and then searching the actual files — provides great flexibility, in particular it allows approximate matching and regular expressions, as we will discuss later.

We have been using *glimpse* for several months and we are finding it indispensable. It is so convenient that we use it many times even when we know where the information is and can use *agrep* directly: It is just easier to quickly browse through the output than to `cd` to the right directory, `ls` to find the exact name of the file, *agrep* it, and `cd` back. We found ideas that we wrote many years ago, and not only forgot where we put them, but forgot about writing them in the first place. We found that we sometimes save information that we probably would have discarded without *glimpse*, because we now have some confidence of ever finding it again. An interesting example, conveyed to us by someone who tested a beta version of *glimpse*, was to find an e-mail address of a friend who moved recently; *glimpse* found it not in any mail messages, but in a call for papers! *Glimpse* can sometimes truly find a needle in a haystack.

2. The Two-Level Query Approach

In this section, we describe our scheme for two-level indexing and searching. We start with the way the index is built.

The information space is assumed to be a collection of unstructured text files. A text consists of a sequence of *words*, separated by the usual delimiters (e.g., space, end-of-line, period, comma). The first part of the indexing process is to divide the whole collection into smaller pieces, which we call *blocks*. We try to divide evenly so that all blocks have approximately the same size, but this is not essential. The only constraint we impose is that the number of blocks does not exceed $2^8 = 256$, because that allows us to address a block with 8 bits (one byte). This is not essential, but it appears to be a good design decision.

We scan the whole collection, word by word, and build an index that is similar in nature to a regular inverted index with one notable exception. In a regular inverted index, every occurrence of every word is indexed with a pointer to the exact location of the occurrence. In our scheme every word is indexed, but not every occurrence. Each entry in the index contains a word and the block numbers in which that word occurs.

Even if a word appears many times in one block, only the block number appears in the index and only once. Since each block can be identified with one byte, and many occurrences of the same word are combined in the index into one entry, the index is typically quite small. Full inverted indexes must allocate at least one word (4 bytes), and usually slightly more, for each occurrence of each word. Therefore, the size of an inverted index is comparable to the size of the text. But our index contains only the list of all unique words followed by the list of blocks — one byte for each — containing each word. For natural language texts, the total number of unique words is usually not too large, regardless of the size of the text.

The search routine consists of two phases. First we search the index for a list of all blocks that may contain a match to the query. Then, we search each such block separately. Even though the first phase can be done by hashing or B-trees, we prefer sequential search using *agrep*, because of its flexibility. With hashing or B-trees, only keywords that were selected to be included in the data structure can be used. With sequential search we can get the full power of *agrep*. Since the index is quite small, we can afford sequential search.²

Agrep is similar in use to other *grep*'s, but it is much more general. It can search for patterns allowing a specified number of errors which can be insertions, deletions, or substitutions (or any combination); it can output user-defined records (e.g., paragraphs or mail messages), rather than just lines; it supports Boolean queries, wild cards, regular expressions, and many other options. Given a pattern, we first use *agrep* to find all the words in the index that match it. Then, using *agrep* again, we search the corresponding blocks to find the particular matches. The same procedure holds for all types of complicated patterns such as ones that contain wildcards (e.g., U.nix), a set of characters (e.g. count[A-E]to[W-Z], where [A-E] stands for A, B, C, D, or E), or even a negation (e.g., U[!l].ix, where [!l] stands for any character except for l). These kinds of patterns cannot be supported by the regular hashing scheme that looks up a keyword in the table, because such patterns can correspond to hundreds or even thousands of possible keywords.

Boolean queries are performed in a similar way to the regular inverted lists algorithm. Suppose that the query is for *pattern1* AND *pattern2*. We find the list of all blocks containing each pattern and intersect them. Then we search the blocks in the intersection. Notice that even if we find some blocks that contain *pattern1* and *pattern2*, it does not mean that the query is successful, because *pattern1* may be in one part of the block and *pattern2* in another. *Glimpse* is not efficient for Boolean queries that contain very common words. The worst example of this weakness that we encountered was a search for "linear programming." This term appeared in our files in several blocks, but to find it we had to intersect all blocks that contain the word 'linear' with all blocks that contain the word 'programming' which in both cases were almost all blocks.

The idea of using *agrep* to search the index can also be integrated with regular inverted indexes. It is possible to separate the list of words in an inverted index from the rest of the index, then use *agrep* to find the words that match the query (e.g., a query that allows some errors), then use the regular inverted index algorithm to find those words. We are not familiar with any system that provides approximate matching in this fashion.

In summary, we list the strengths and weaknesses of our two-level scheme compared with regular inverted indexes:

² Although the index is not an ASCII file, because the block numbers use all 256 byte values, the words themselves are stored in ASCII so *agrep* can find them.

Strengths

1. Very small index.
2. Approximate matching is supported.
3. Fast index construction.
4. No need to define document boundaries ahead of time. It can be done at query time.
5. Easy to customize to user preferences.
6. Easy to adapt to a parallel computer (different blocks can be searched by different processors).
7. Easy to modify the index due to its small size. Therefore, dynamic texts can be supported.
8. No need to extract stems. Subword queries are supported automatically (even subwords that appear in the middle of words).
9. Queries with wildcards, classes of characters, and even regular expressions are supported.

Weaknesses

1. Slower compared to inverted indexes for some queries. Not suitable for applications where speed is the predominant concern.
2. Too slow, at this stage (but we're working on it), for very large texts (more than 500MB).
3. Boolean queries containing common words are slow.

3. Usage and Experience

We have been using *glimpse* for a few months now, mostly on our own file systems. We have tried it on several other data collections, ranging in size from 30MB to 250MB. In this section, we discuss some of the current features of *glimpse* and our experience with it.

The current version of *glimpse* is a collection of many programs of about 7500 lines of C code. It is geared towards indexing a part of a file system. The statement

```
glimpse_index [-n] dir_name [dir_names]
```

indexes directory *dir_name* (or several directories) and everything below it. The most common usage is `glimpse_index ~`. The `-n` option indexes numbers as well. Indexing is typically done every night, and it takes about 6 minutes (elapsed time) to index 50MB of text or about 7-8 seconds per 1MB (using a DEC 5000/240 workstation).

Before indexing a file, the program checks whether it is a text file. If the file is found to have too many non-alphanumeric characters (e.g., an executable or a compressed file), it is not indexed, and its name is added to a `.log` file. Other formats, for example, 'uuencoded' files and 'binhexed' files, are excluded too. Determining whether a file is a text file is not easy. Texts of languages that use special symbols, such as an umlaut, may look like binary files. There are tools that do a good job of identifying file types, for example *Essence* [HS93], and we will eventually incorporate them in *glimpse*. On the other hand, some ASCII files should not be indexed. A good example is a file containing DNA sequences. We actually have such files and found them to cause the index to grow significantly, because they contain a large number of essentially random strings. We should note that the two-level scheme is not suitable for typical biological searches, because they require more complicated types of

approximate matching. Indexing numbers is useful for dates and other identifying numbers (e.g., find all e-mail messages from a certain person during July 1991). But large files with numeric data will make the index unnecessarily large. *Glimpse* allows the user to specify which files should not be indexed by adding their names to a `.prohibit` file. We plan to add more customized features to the indexing process.

The partition into blocks is currently done in a straightforward way. The total size of all text files is computed, and an estimate on the desired size of a block is derived. Files are then combined until they reach that size, and a new block is started. We plan to improve this scheme in the future in two ways: 1) the partition should be better adapted to the original organization of the data, and 2) the user should have the ability to control how the partition is done.

The user interface for *glimpse* is similar to that of *agrep*, except that file names are not required. The typical usage is

```
glimpse information
```

which will output all lines in all indexed files that contain *information*;

```
glimpse -l HardToSpell
```

will find all occurrences of *HardToSpell* with one spelling error.

```
glimpse -w 't[a-z]j@#uk'
```

will find all email addresses in which a 3-letter login name starts with *t* and ends with *j* (it is typical not to know the user's middle name) and *uk* is somewhere in the host name (in *agrep* the symbol *#* stands for arbitrary number of wild cards, and the *-w* option specifies that the pattern must match a complete word).

Glimpse supports Boolean queries the same way that *agrep* does (with *;* serving as AND):

```
glimpse -l 'Lazoska;Zahorian'
```

will find all occurrences of both names (allowing one spelling error, which is needed).

```
glimpse 'Winter Usenix'
```

will first search the index for the two words separately and find all the blocks containing both words, then use *agrep* for the whole phrase.

Another nice feature of *glimpse* is to limit the search to files whose names match a given regular expression.

```
glimpse -F 'haystack\.c$' needle
```

will find all needles in all haystack.c files. Sometimes, you want to search everywhere for that elusive definition:

```
glimpse -F '\.h$' ABC_XYZ
```

will search all .h files.

```
glimpse -F '\.tex$' 'environment;^\\"'
```

will find all occurrences of *environment* in a line that starts with ** (useful if you want to see *environment* as part of a definition). *Glimpse* is in fact so flexible that it can be used for some file management operations; for example,

```
glimpse -F mbox -h -G . > MBOX
```


will concatenate all files whose name is mbox into one big one;

```
glimpse -c -F 'pattern\.h$' '\#define'
```

will provide a list of all pattern.h files, each followed by the number of #define's it has. We even allow errors in the regular expressions for the file names, using, for example, -F1 for one error.

4. Performance

All the performance numbers given below are anecdotal. The point of this section is not to compare the running time of *glimpse* to other systems, but to argue that it is fast enough for its purposes. Query times depend heavily on the type of indexed data and the query itself (not to mention the architecture and the operating system). If the pattern appears in only one block, the search time will be almost always fast no matter how large the data is or how complex the pattern may be (an unsuccessful query is the fastest because only the index needs to be searched). If the pattern matches a large portion of the blocks, the query may be slow. Searching in a large information space requires a careful design of queries. No matter how fast the search is, if the number of matches is large, it will take too long to sift through them. However, we found that even matching to common patterns can be useful, because one starts to see many matchings quite fast, and one can then stop and adjust the pattern. In some cases, even though there were many matches, the first few ones were sufficient to get the answer we were looking for. (We also added an option -N that tells the user the number of blocks that would need to be searched.)

Below we list some running times (on a DEC 5000/240 workstation) for a personal file system containing 69MB of text in 4296 different files. It took 4.9 minutes of CPU time (9 minutes of elapsed time) to index it (including the time to determine that 857 other files were not text files) and the index size was 1.9MB, which is 2.7% of the total. 205 blocks were used. A typical search takes from 2-10 seconds. For example, a search for *biometrics* took 1.6 seconds. (The numbers listed here are the sum of the user and system times.) A search for *incredible* took 2.8 seconds (there were 27 matches in 22 files). A search for *Czechoslovakia* allowing two errors took 3.6 seconds. A search for *Usenix* took 13.9 seconds because there were 93 matches divided among 70 different blocks. A Boolean search can sometimes help by limiting the number of scanned blocks (which are only those that match all patterns), but again it depends on the number of matches. For example, searching for *Usenix* AND *Winter* took only 4.5 seconds since there were only 19 matches. A search for *Lazowska* AND *Zahorjan* with one error took 4.7 seconds. A search for *protein* AND *matching* took 7.7 seconds because both patterns were very common. The search for the e-mail address mentioned above (*t[a-z]j@uk*) took 30 seconds (although we believe that some of it is due to a bug that for some complicated regular expressions causes search in some unnecessary blocks), but this is still much better than any alternative.

For a much larger information space, we indexed an old copy of the entire library catalog of the University of Arizona. This experiment and the next one were run on a SUN SparcStation 10 model 512 running Solaris. The copy we used lists approximately 2.5 million volumes, divided among 440 files occupying 258MB. The index size was 8MB, which is 3.1%. The index would have been much smaller for regular text; the catalog contains mostly names and terms, in several languages, and since it uses fixed records, quite a few of the words are truncated. It took 18 minutes (all times here are elapsed times) to index the whole catalog. A search for *Manber* took 1 second (one match). A case insensitive (-i) search for *usenix* yielded one match in 1.8 seconds. A search for *UNIX* took 3.9 seconds (138 matches in 8 blocks). A search for *information* AND *retrieval* took 14.9 seconds; there were 38 matches in 9 blocks, but 31 blocks had to be searched (because they included both terms).

Searching for 'algorithm' allowing two errors took 16.3 seconds, finding 269 matches, 21 of which did not match the pattern exactly due to a misspelling (algorithn, Alogrithms) or a foreign spelling (e.g., algoritmy).

An example that highlighted the "best case" for *glimpse* was a directory containing the archives from comp.dcom.telecom. They occupy about 71MB divided among 120 files. The small number of files minimizes the overhead of opening and closing files making *agrep*, and as a result *glimpse* very fast. Even without *glimpse*, searching for Schwarzkopf allowing two misspelling errors took *agrep* only 8.5 seconds elapsed time. It took *glimpse* 3:38 minutes to index the directory and the index size was 1.4MB (about 2% of the total). With *glimpse*, the Schwarzkopf search took 0.4 seconds (all matches were in one file). A search for Usenix (no error allowed), which appeared in 5 blocks, took 0.9 seconds. A search for *glimpse* took 3.7 seconds (15 matches in 9 blocks).

5. Future Work

We list here briefly some avenues that we are currently exploring.

5.1. Searching Compressed Text

If the text is kept in a compressed form, it will have to be decompressed before the sequential search can be performed. This will generally slow down the search considerably. But we are developing new text compression algorithms that actually *speed up* the search while allowing compression at the same time. The first algorithm [Ma93] allows *agrep* (and most other sequential search algorithms) to search the compressed file directly without having to decompress it first. Essentially, instead of restoring the text back to its uncompressed form, we modify the *pattern* to fit the compressed form and search for the modified pattern directly. Searching the compressed file directly improves the search time (as well as space utilization, of course), because there is less I/O. In preliminary tests, we achieved about 30% reduction in space and 25% decrease in search time. This allows files to be kept in a compressed form indefinitely (unless they are changed) and to be searched at the same time. In particular, we intend to compress the index used in the two-level scheme, because it is searched frequently. We are working on another compression scheme that will be integrated into *glimpse*. This work is still in a very preliminary stage. The compression rates, for natural language texts, seem to be in the 50% range. It also allows fast search without decompression, although it is too early to predict its speed.

5.2. Incremental Indexes

The two-level index is easier to modify and adapt to a dynamic environment than a regular inverted index, because it is so much smaller. To add a new file to the current index, we first add the file to an existing block or create a new block if the file is large enough or important enough to deserve it. Then we scan the index and add each word in the new file that does not already appear in the block. Since the index is small, reading it from disk and writing it back can be done very fast. Deletion of a file is slightly more time consuming because for each word we need to check the whole block to determine whether that word appears somewhere else (and thus should not be deleted). Fortunately, *agrep* contains a very powerful algorithm for multi-pattern matching. It can search for a large collection of words (up to 30,000) concurrently.

Incremental indexing will be essential for indexing newfeed. We are considering adapting *glimpse* to allow searching usenet netnews. Currently, the total size of a typical usenet server is from 500-800MB. Quite a bit of it is not text but images and programs, so it is a size we can handle. The problem is that this kind of newfeed consists of a large number of small files (individual email messages) stored at random places on the

disk. A better data organization will probably be required for reasonable performance.

5.3. Customization

There are several ways to let the user improve the indexing and searching procedures by customizing. The user should be able to decide whether certain patterns are indexed or not. We provide an option to index numbers, but we should allow more flexibility. One way is to provide a hook to an external filtering program, provided by the user, which will decide what to index based on contents, type of file, name of file, etc. We also plan to allow the user to store a large set of multi-words phrases (e.g., "fiber optics," "Jonathan Smith," "May 1992") that will be indexed together, allowing quick search for them without the need for Boolean queries. Another option is to partition the collection of files into categories and build separate indexes for each (e.g., correspondence, information from servers, program source codes). This is not needed for small to medium size file system, but may be essential for large file systems. We also plan to support access to any special structure or additional information associated with the text. For example, some searches may specify that the desired information starts at column 30 on the line or in the second field. The user may want to search only small files (say, below 20KB) or recent files (say, after August 1992).

There should be more ways to view the output. For example, for queries that give many matches, the user may be interested first in a rough idea of where those matches are (e.g., only directory names). We currently have an option [-c] to list only the files containing a match along with the number of matches, and another option [-N] to output only the number of blocks with potential matches. These options, and many more, can be incorporated in *glimpse* rather easily. We believe that customization is the key to better search facilities.

Acknowledgements

Burra Gopal found and corrected several bugs in the code. We thank Greg Andrews, George Forman, and Trevor Jenkins for comments that improved the manuscript.

References

[BN91]

Bradford R., and T. Nartker, "Error correlation in contemporary OCR systems," *Proc. of the First Int. Conf. on Document Analysis and Recognition*, (1991), pp. 516-523.

[Fa85]

Faloutsos C., "Access methods for text," *ACM Computing Surveys*, 17 (March 1985), pp. 49-74.

[GB91]

Gonnet, G. H. and R. A. Baeza-Yates, *Handbook of Algorithms and Data Structures* (Chap. 7.2.6) Second Edition, Addison-Wesley, Reading, MA, 1991.

[HS93]

Hardy D. R., and M. F. Schwartz, "Essence: A resource discovery system based on semantic file indexing," *Proc. of the USENIX Winter Conference*, San Diego (January 1993), pp. 361-374.

[Ma93]

Manber U., "A text compression scheme that allows fast searching directly in the compressed file," technical report 93-07, Department of Computer Science, University of Arizona (March 1993).

[RKN92]

Rice, S. V., J. Kanai, and T. A. Nartker, "A report on the accuracy of OCR devices," Technical Report, Information Science Research Institute, University of Nevada, Las Vegas, 1992.

[SM83]

Salton G., and M. J. McGill, *Introduction to Modern Information Retrieval*, McGraw-Hill, New York, 1983.

[Te82]

Teskey, F. N., *Principle of Text Processing*, Ellis Horwood Pub., 1982.

[WM92a]

Wu S. and U. Manber, "Agrep — A Fast Approximate Pattern-Matching Tool," *Usenix Winter 1992 Technical Conference*, San Francisco (January 1992), pp. 153–162.

[WM92b]

Wu S., and U. Manber, "Fast Text Searching Allowing Errors," *Communications of the ACM* **35** (October 1992), pp. 83–91.

Biographical Sketches

Udi Manber is a professor of computer science at the University of Arizona, where he has been since 1987. He received his Ph.D. degree in computer science from the University of Washington in 1982. His research interests include design of algorithms, pattern matching, computer networks, and software tools. He is the author of "Introduction to Algorithms - A Creative Approach" (Addison-Wesley, 1989), and the editor of 3 other books. He received a Presidential Young Investigator Award in 1985.

Sun Wu is an associate professor of computer science at the National Chung-Cheng University, Taiwan. His research interests includes distributed systems, software tools, and design of algorithms.

Drinking from the Firehose: Multicast USENET News

Paper Author: Kurt Lidl
Code Author: Josh Osborne
Code Author: Joseph Malcolm

News transport and spooling systems of the last several years have concentrated on decreasing the resource load on news servers. One beneficial side effect has been the average decrease in time that a news system spends on a given article. This paper describes a novel USENET news transport protocol, which we call *Muse*. The two major motivations behind *Muse* are to reduce the average propagation delays of articles on USENET and to further reduce the resource load on a centralized news server. *Muse* runs on top of the experimental Internet multicast backbone, commonly referred to as the *Mbone*. Major design and implementation issues are discussed. Security concerns of multicast news are discussed and our solution is examined. The problems of scaling news distribution to thousands of hosts are also addressed.

1. Introduction

USENET is a distributed messaging system, which is implemented as a software system layer over a variety of networks and connections. The basic message unit is the *article*. Articles are logically grouped into *newsgroups*. The format of each article is described in [Adams87].

One of the primary concepts of the USENET "network" is that of a *news-neighbor*. A site's news-neighbors are defined as those sites that have bidirectional transfers of articles with a given site. Each news site has a unique identifier, which is known as that site's *news-name*. All traditional news systems know the *news-name* for all their news-neighbors. Each USENET site that generates an article passes that article to its news-neighbors, based on several control criteria. These criteria are based on which newsgroup an article has been posted to, what the distribution of the article is and whether or not the news-neighbor has seen the article before.

The *Path:* header of each article is one of the mechanisms used to prevent message loops from occurring between news sites. The loop prevention logic embodied by the *Path:* header is executed on the news system that is sending an article. Before sending an article to its news neighbors, each site prepends its *news-name* to the *Path:* header. News systems are required to check the *Path:* header against the news-name of the candidate receiving site. If the news-name of the receiving site under consideration is already in the *Path:* header, the article has already passed through the site and does not need to be queued for transit there. Thus, the sending news site can safely not transfer the message to that receiving site.

The second form of duplication suppression is implemented by the receiving news site, in the form of *message-ID* checking. Each news site keeps a database of the *message-IDs* of all the articles that have been received by that system. These records are normally kept for two to three weeks and then purged from the database. When a news site receives an article or is offered an article via NNTP, it checks for the existence of the *message-ID* in the database. If the site finds the *message-ID* already in the database, the news system has already received the message and can safely ignore the duplicate copy of the article. For large news systems with many news-neighbors, connected via high speed communications lines, the resources spent doing duplicate suppression can be very significant. Doing the *message-IDs* check is the equivalent of a database search operation, which must be performed at peak rates of over two hundred times a second, on busy news servers.

Traditionally, there have been two major news transfer methods – UUCP spooling and NNTP. NNTP is the newer of the transfer protocols and features a SMTP-like dialogue for each article that is sent. It is worthwhile to note that NNTP requires a TCP connection between the sender and the receiver of the news. UUCP spooling consists of writing multiple news articles to a single batchfile and then using *uucp* to copy

this file to the receiving system, where the news is processed. UUCP does not require a TCP connection and can be configured to run over a wide variety of connections, including dialup modems, X.25 networks, directly connected serial ports and TCP/IP networks.

2. Motivations behind Multicast USENET News

The major design goal for *Muse* is to significantly reduce the average article propagation delay through USENET. This decrease in propagation time is beneficial to the users of USENET, as it changes the perception of the network making it seem that it is more responsive to questions and answers. Additionally, the decrease in propagation latency of articles has the effect of smoothly distributing a graph of the traffic flowing over backbone network links.

Muse was carefully designed so that an unlimited number of news listeners can be supported from a single multicast news server. News distribution via *Muse* should scale without limit, as the total number of listeners is bound only by the size of the *Mbone*, and not by exhaustion of centralized resources [Salz92], as is the case in all traditional news systems. Each additional listener consumes only the local resources of the listener (e.g. network bandwidth, disk I/O, CPU cycles). In all cases, the same amount of effort is expended on the *Muse* transmitter for no *Muse* listeners as for a thousand *Muse* listeners.

Muse was conceived in such a way that widespread deployment of this software should actually decrease the amount of traffic due to USENET news that travels various backbone connections. Since *Muse* performs a flood-fill of a single news article over a significant portion of the Internet, only one copy of the article should traverse most backbone links. Common sense dictates that it should be much more efficient to implement a broadcast mechanism via a multicast network, rather than modeling the broadcasts via many unicast network links. USENET, after all, is a broadcast network that is currently implemented over many unicast links.

A final motivation of the authors was the hope that by basing *Muse* on the *Mbone*, it would encourage others in the networking community to write innovative software that addresses old problems in a new fashion. By basing our transport system on a multicast transport, we wish to give another reason to stabilize the *Mbone* so that it will be more reliable in the future and other production services might be offered on top of it.

3. General Architectural Overview

Muse is a USENET news transport layer that is implemented in two logical parts. The first part is the news transmitter, which accepts news articles, checksums the article, digitally signs the checksum using the RSA algorithm and multicasts them. The listener receives the multicast, checksums the article independently and verifies the checksum against the encoded checksum included with the article.

4. Introduction to Technologies Used

Muse uses several technologies that are not yet in widespread use across the Internet. Because these technologies are not all in day-to-day use across the Internet, many people are unfamiliar with them. A short description and explanation of these technologies is contained in this section of the paper.

4.1 The Mbone

The *Mbone* is the experimental multicast backbone that various network service providers, universities, research institutions, and corporations have layered on top of the Internet. It can best be described as a large distributed experiment in multicasting. To better understand the *Mbone* it is important to understand the basics of multicasting. The following description of multicasting datagrams is taken from [Deering89].

IP multicasting is the transmission of an IP datagram to a "host group" a set of zero or more hosts identified by a single IP destination address. A multicast datagram is delivered to all members of its destination host group with the same "best-efforts" reliability as regular unicast IP datagrams, i.e., the datagram is not guaranteed to arrive intact at all members of the destination group or in the same order relative to other datagrams.

The membership of a host group is dynamic; that is, hosts may join and leave groups at any time. There is no restriction on the location or number of members in a host group. A host may be a member of more than one group at a time. A host need not be a member of a group to send datagrams to it.

A host group may be permanent or transient. A permanent group has a well-known, administratively assigned IP address. It is the address, not the membership of the group, that is permanent; at any time a permanent group may have any number of members, even zero. Those IP multicast addresses that are not reserved for permanent groups are available for dynamic assignment to transient groups which exist only as long as they have members.

4.2 MD5 – Message Digest checksum

In the *Muse* system, we have a need for a fast, cryptographically secure checksum algorithm. We selected the MD5 Message Digest algorithm. The following description of the MD5 article checksum is taken from [Kaliski93].

A message-digest algorithm maps a message of arbitrary length to a “digest” of fixed length, and has three properties: Computing the digest is easy, finding a message with a given digest—“inversion”—is hard, and finding two messages with the same digest—“collision”—is also hard. Message-digest algorithms have many applications, including digital signatures and message authentication.

RSA Data Security’s MD-5 message-digest algorithm, developed by Ron Rivest [Rivest92], maps a message to a 128-bit message digest. Computing the digest of a one megabyte message takes as little as a second. While no message-digest algorithm can yet be *proved* secure, MD5 is believed to be at least as good as any other that maps to a 128-bit digest. Inversion should take about 2^{128} operations, and collision should take about 2^{64} operations. No one has found a faster approach to inversion or collision. [Robshaw93]

4.3 Public key cryptography systems

The *Muse* system requires a method of distributing the message-digest for each article in a form that may be deciphered by the *Muse* listeners, but may not be spoofed by illegitimate multicast transmitters. The RSA public key cryptography system was chosen as the public key system for use, mainly due to its freely available (in the U.S.A.) reference implementation. The following description of how public key cryptography systems is taken from [Fahn93].

Traditional cryptography is based on the sender and receiver of a message knowing and using the same secret key: the sender uses the secret key to encrypt the message, and the receiver uses the same secret key to decrypt the message. This method is known as secret-key cryptography. Public-key cryptography was invented in 1976 by Whitfield Diffie and Martin Hellman in order to solve the key management problem. In the new system, each person gets a pair of keys, called the public key and the private key. Each person’s public key is published while the private key is kept secret. The need for sender and receiver to share secret information is eliminated: all communications involve only public keys, and no private key is ever transmitted or shared.

4.4 The NNTP Protocol

The *Network News Transfer Protocol* as described in [Kantor86], provides a news articles transfer protocol that runs on top of a reliable stream connection, such as TCP. The protocol’s sample implementation was originally released as the software package called “*nntp*”. This software package has been widely ported and is available on a large number of systems. NNTP has a well defined protocol command set, response codes and transfer mechanism for news articles. Additionally, NNTP has the distinction of having been independently implemented several times. For IP connected sites, NNTP is the preferred mechanism for transferring USENET news.

The NNTP "IHAVE" command specifies the start of a command sequence that a news server and client typically exchange. The news server will assert "IHAVE <message-id>" for a given message. The receiving NNTP host will respond "335 send article," which indicates that it has not seen that particular message before and desires a copy of it. Or, the receiver will send "435 article not wanted," which typically indicates that the receiver has already gotten a copy of that article and does not need a duplicate copy.

5. Muse: Overview of Operation

Muse operates in a classical transmitter-listener scenario. The reason that *Muse* can multicast to an unlimited number of machines is that *Muse* requires that *no* packets are sent from the listener back to the multicast transmitter. As a consequence of this, it is easy to break the the examination of the transmitter and the listener into two discrete parts. In fact, the *Muse* system currently consists of two programs, *n2m*, which implements the transmitter functions, and *m2n*, which implements the listener functions.

Muse's transmitter accepts the article to be multicast through a standard NNTP "IHAVE" protocol transaction. This approach was taken to minimize the the integration effort required to make *Muse* talk to existing news systems. Because of this standard interface to other news systems, *Muse* just looks like another news feed. As such, all the customary NNTP news feed controls available on a NNTP server may be applied to the *Muse* newsfeed.

Once *Muse* has received the article, it checks to see if *Muse's* news-name already appears in the *Path:* header. The news system that is sending articles to the *Muse* transmitter should not attempt to feed any such articles to *Muse*. The extra check is fairly cheap and prevents a potential waste of bandwidth. *Muse* follows the guidelines listed in RFC1036 [Adams87] for parsing the *Path:* header. If it finds its news-name already in the header, it will drop the article. This is done to ensure that a given *Muse* transmitter will not multicast an article a second time. If *Muse* does not find its name, it adds its news-name to the *Path:* header, fills in the rest of the protocol header fields and calculates the MD5 checksum of the article. Once the *Muse* listener has the MD5 checksum, it signs the checksum using its private key. The complete *Muse*-modified article is then ready to be multicast to the *Muse* listeners. The transmitter will send the *Muse* article to the appropriate pre-arranged multicast group as a single (possibly large) UDP packet. In its present form, *Muse* depends on the operating system to fragment the UDP packet correctly to ensure transmission across any networks.

After being transmitted, the news article (along with the *Muse* header), is in transit on the *Mbone*. The articles representation is one or more IP packets, representing the entire UDP packet. The *Mbone* will deliver this packet stream to those machines that are interested in receiving the news multicast.

It is important to keep in mind that the receiving and verifying stage of the *Muse* transport system happens more or less simultaneously on all *Muse* listeners across the *Mbone*. Thus, each listener will go through this process.

The *Muse* multicast listener (*m2n*) performs work according to the following outline. The first job is to alert the operating system kernel to its desire to listen on the correct (pre-arranged) multicast channel. The operating system kernel is relied upon to defragment the collected IP fragments into a complete UDP packet. If not all the fragments of the original UDP packet make it to the destination host, the operating system kernel of the destination will drop the remaining packets of the article.

Once the entire *Muse* article has been received, a combination of the received key number and the last news-name in the *Path:* header is used to look up the appropriate public key in the local matrix of public keys. If no key is found for the sending news host, the article is dropped and the condition is logged to a file on disk. Assuming an appropriate public key is found for the transmitting news site, the listener calculates a local version of the MD5 checksum for the article as it was received. Then, the listener extracts the received MD5 checksum from the *Muse* header, and compares it to the locally generated checksum. If the comparison shows a difference, the article is dropped, as one of the following invalidating actions has occurred: the article was mangled or modified in transit, the article was sent by a multicast server that was spoofing an authentic *Muse* server, or the public key used to decrypt the MD5 checksum was invalid. If the article was rejected for any of these reasons, a timestamp and message-ID (if one can be recovered from the article) are logged to a file on disk, to assist in problem tracking. Assuming the checksums matched the received and

verified article is then sent via a NNTP "IHAVE" transaction to the receiving NNTP server. Note that the receiving NNTP host does not need to be the host running the *Muse* listener software.

Muse provides a mechanism for sending various control messages across the *Mbone* to all running *Muse* listeners. These control messages are intended for collecting listener audience sizes and statistics gathering. *Muse* has implemented a *Version* control message, which is similar in spirit to the more traditional USENET *Version* control message. In addition to the statistics gathering message types, a *Revoke* message has been implemented, to partially address the issue of key management for the public keys that *Muse* uses. If a *Revoke* control message is received, the public key that was used to sign the message is permanently revoked and should be removed from use. Another control message that has been implemented for this release of *Muse* is the *NOP* (no operation). This message is broadcast once every 60 seconds, if no other news articles have been sent during that time period. This is intended as a *Muse* keepalive mechanism, to be used in conjunction with the results of research being conducted into how to best handle failure of the primary NNTP news feed into the *Muse* transmitter. The final control message implemented is the *News* control message. It is the "standard" type of packet sent by the *Muse* transmitter and is used to indicate a news article is contained in the packet.

6. *Muse*: Justification of Design Decisions

- Why use UDP at the transport layer?

UDP was chosen as the transmission method for two major reasons. First, since it is a datagram based protocol, UDP maps closely to the type of traffic that characterizes USENET news. This type of traffic could be characterized as having many totally independent packets, with no retained state across any of the packet flows. This is very important if one looks at the scaling issues of running a global news system. As UDP does not require any acknowledgment of received packets, it is ideal for building an application that will scale without bound. Second, UDP was chosen because there is no need to retain any state across the server and the listeners. This stateless nature makes *Muse* relatively maintenance free in the face of network outages and server crashes.

The simplicity of the UDP model was the reason that we choose to allow IP fragmentation for breaking apart large articles, as opposed to coding our own article fragmentation system. While relying on IP fragmentation, which limits *Muse* to a theoretical maximum 64 kilobyte sized packet and an operating system imposed 9 kilobyte article size limit (in both cases, minus the *Muse* protocol overhead), analysis has shown that 9 kilobyte packets are sufficient to transfer approximately 96% of all news articles.¹

- Why is fragmentation left to the UDP layer in the kernel?

The decision to rely on UDP fragmentation does not close the door on implementing a different fragmentation scheme for later releases of the software. The option of coding this for the initial release was deemed unnecessary, as the stability of the *Mbone* also comes into play. The *Mbone*, as it exists currently,² sometimes exhibits severe packet loss during high network loads. The probability of receiving 64 kilobytes of data with no fragments missing was deemed unlikely. We are content to accept the artificially low limit of 9000 byte articles in the initial release of the software. While it would be possible to retransmit these articles, the *Muse* transmitter has no way of knowing that the article was dropped in transit, nor does it have a way of "replaying" a given article, as it retains no state.

We are examining other work that has been performed in this area, notably the *Coherent File Distribution Protocol*. [Ioannidis91] Further discussion of this topic is in the *Future Work* section of this paper.

- When selecting a public key cryptography system, why use RSA instead of NIST's Digital Signature Standard (DSS)?

The decision of choosing RSA's public key cryptography system over NIST's was influenced by the following reasons. There was a freely available RSA reference implementation. There is not, to the best of

¹ Based on traffic analysis performed on two weeks of articles passing through the news site *darwin.sura.net*, during February and March, 1993.

² July–October, 1993

our knowledge, a similar reference implementation for the DSS. The other mitigating factor was that the RSA algorithm takes a long time to sign each checksum, but the decode of the checksum is relatively quick. With the DSS, the opposite is true – the initial signing of the checksum would be quick, and the verification stage would be slow, due to high CPU usage.

- Why do we need to RSA sign the articles at all?

To the casual observer of USENET news, it would not seem necessary to be concerned with the problems of security along this transport path. However, we contend that security of this transport is paramount to *Muse*'s success. What made the authors of *Muse* so security conscious? Because the transmitter and listeners collect no state during normal news transmission, it would be easy for anyone to inject a forgery into the news stream, simply by source-routing an "article" to a site that is known to be collecting news via the *Muse* multicast. This packet could have had all the incriminating data in it removed – e.g. the source IP address could be fake or the *Path:* header could be wrong. There would be no method to determine if an article was faked without the digital signature, leaving the listener no option but to believe the received article is valid.

This would have severe repercussions to the news community. Since the goal of the *Muse* software is to make the transport of news nearly instantaneous from coast-to-coast, it raises the possibility of USENET terrorism to new heights. Not only could people receive articles very quickly, but they could cancel them just as quickly. Often, this would be before another human ever saw the article! Silencing an unruly upstart on a newsgroup would be a trivial matter of coding and would be entirely untraceable, without the digital signing of the articles.

The traditional USENET news article has a *Path:* header that provides a trace of which news sites an article has traveled through. While this trace is not completely trustworthy in the case of forged articles, it is possible to track down where a forged news article was injected with the assistance of the news administrators along the way. This is mainly due to logging of the sites to which a news system forwards a copy of a given article. With multicast IP traffic, the situation is considerably more difficult. In fact, this problem is intractable with the current multicast system. Just as no IP routers can effectively track all packets sent through them on a packet by packet basis, no multicast router can log all packets traversing that machine. With traditional news systems, only connections and connection summaries need to be logged to trace the flow of news. With *Muse*, the headers of every packet would need to be saved in order to provide a comparable level of logging to allow tracking of which articles came from where. Because the source address on UDP packets is nearly worthless as a security check, a different authentication mechanism needed to be deployed, one where the identity of the *Muse* transmitter could be verified by the listener.

- Why have it use NNTP, rather than designing something else to take its place?

NNTP is a well established protocol, with many robust implementations in existence and use. Using NNTP as both the initial acceptance mechanism for the newfeed and the final presentation protocol allows rapid widespread deployment across the Internet. Since the *Mbone* is not widely available currently, it was deemed useful to be able to receive articles on an *Mbone*-reachable machine and direct the resulting NNTP feed to a host that is not necessarily directly on the *Mbone*.

- Why does *Muse* have a news-name when it is not a complete news system in the traditional sense?

We chose to give the *Muse* transmitter a news-name, even though it is not a complete news system in the traditional sense. This was done to allow statistics gathering on articles that are multicast and accepted by other news systems. This will allow immediate generation of "news influence" ratings, as the software that generates these statistics is already in place across the USENET backbone.

The second reason for adding our own news-name to the article is to avoid a duplicate multicast of the article in the case that the transmitter is sent an article a second time. This will become more important in the future, when the transmitter host has the ability to accept more than a single incoming news-feed.

- How is access control implemented in the *Muse* transmitter?

Quite frankly, there is currently no access control mechanism built into the *Muse* transmitter (*n2m*). We rely on *tcpwrapper* [Venema92] to provide access controls to the machine that runs the *Muse* transmitter. *Tcpwrapper* allows replacement of a network daemon by a stub program, that validates or rejects network connections before actually invoking the real network daemon.

Logically, access control for the *Muse* listener machines is divided into two parts, that of the link between the transmitter and the listener and the link between the listener and the NNTP server that will actually accept the news. The UDP based "connection" from the transmitter to the listener is protected by the RSA signed MD5 checksum on the file. The connection between the *Muse* listener and the NNTP receiver is handled via the traditional access controls to sites running NNTP transport systems.

- How does *Muse* get around the problem of dropped articles and an unreliable transport layer?

In the current implementation, *Muse* does not directly address the case of resending dropped packets in a given article stream. Because *Muse* is not a reliable news transport mechanism, it must be used in conjunction with a backup, fully redundant news transport system, such as a traditional NNTP newsfeed or other backup newsfeed (possibly via an IHAVE/SENDME newsfeed). It is hoped that by adapting basic ideas behind the *Coherent File Distribution Protocol* we can design a way of extending *Muse* to allow resending of dropped article packets. Additionally, implementation of a protocol derived from *Coherent File Distribution Protocol* would provide a method for transmitting articles more than 9000 bytes in length.

Depending on the implementation, adding this code could complicate the *Muse* software greatly. It also has the undesirable effect of limiting, at least to some extent, the number of listeners that can be supported from a single multicast transmitter. This limitation is driven by the need for communication to the transmitter from listeners that have received damaged articles, either due to packet corruption or due to fragment loss.

- How is buffering implemented?

Muse needs no complex buffering system, such as an on-disk article spool that other news systems typically use. (Which, while perhaps not commonly regarded as a "buffer", serves that purpose for article forwarding.) Since the authors of the *Muse* software valued reduced transfer times and ease of administration over reliability of transport, the buffering system that has been developed for *Muse* was both easy to implement and does not require any cleanup after a server crash. The basic idea is to request a fairly large buffer from the kernel (64 kilobytes) on the connection (for both the transmitter and the listener). Since the transmitter is driven by a NNTP server that has already performed all disk I/O, disk latencies are not expected to matter for the transmitter. The transmitter can multicast the article as soon as it receives the article via NNTP and calculates the RSA signature of the MD5 checksum. The kernel will buffer outgoing articles on the transmitter side and multicast them as the ethernet device becomes free.

The *Muse* listener is in much greater need of buffering, since it must offer the articles to other news systems, which must eventually commit the articles to a disk. Traditional news systems almost always have some amount of delay, either due to disk I/O in retrieving the message-id from the *history* file, or in actually committing the article to the disk. During the wait for disk I/O to complete, it is not unreasonable to expect more news to arrive at the listener system, which is what drives the need for effective buffering on the listener. The listener will only remove a single article at a time from the the queue of de-fragmented UDP packets received. The remaining space in the UDP queue will act as a buffer for up to five maximally-sized articles. Because the UDP listener buffer is a queue of items, we can be assured that the oldest articles in the queue will be processed first. Unfortunately, the news articles towards the front of the queue are least likely to be worthwhile processing, as they are older and more likely to be duplicate articles.

7. Performance Numbers and Traffic Analysis

The test data for the following analysis was collected over approximately one month. We have results of our testing from four different news sites, which all entered into the test on different days. Because of this, the number of articles that were sent to each site varies.

Our first graph [Figure 1] shows the total number of articles that we believe were "eligible" to be received by each of the listeners. This number is based on the accounting logs that were generated by each listener. Our method for collection of the statistics gathered the results every eight hours. During each eight hour run, an assumption is made that the listeners are eligible for all articles between the first and the last article appearing in the transmitter's logs that were multicast during that time period. Thus, if a listener first logs reception of a multicast article four hours into a time period, that listener's statistics would at most be accounted for the articles that were multicast during the next four hours of time.

This graph has been included to allow the reader to judge the reliability of the data in the following graphs. All four sites shown have been operating the receiver software long enough to receive at least a thousand articles, and three of them have over three thousand articles. We believe this is a sufficient volume to allow analysis without too much danger of anomalous data coloring the results.

The significant variation in number of articles received is primarily due to the fact that there is a wide difference in the length of time each site has been operating the receiver software. The other factor which is likely to have an effect is the reliability of the news system that is receiving the article. There were also a few problems in the receiver (since believed fixed), which caused the receiver not to run after a reboot until restarted manually.

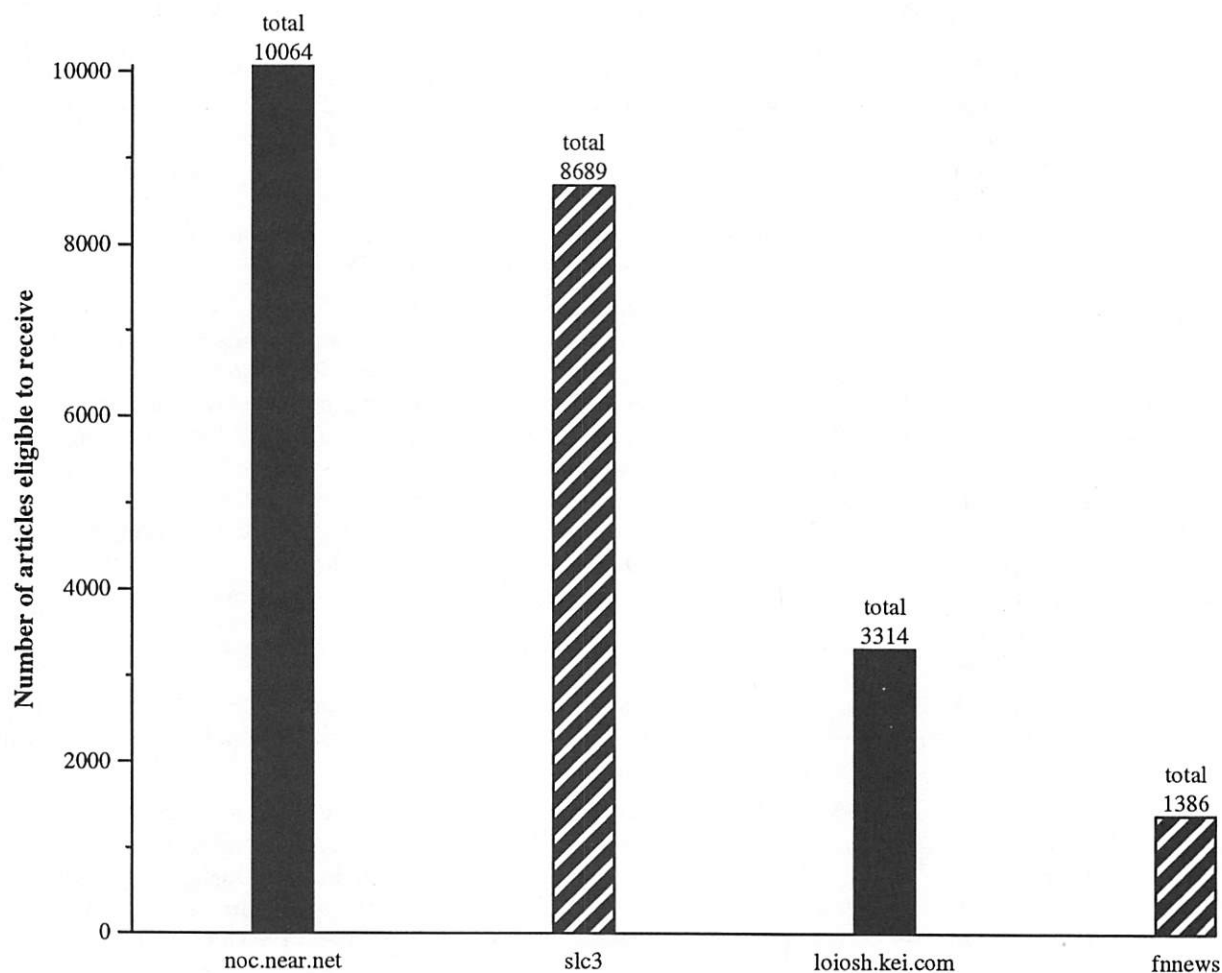


Figure 1

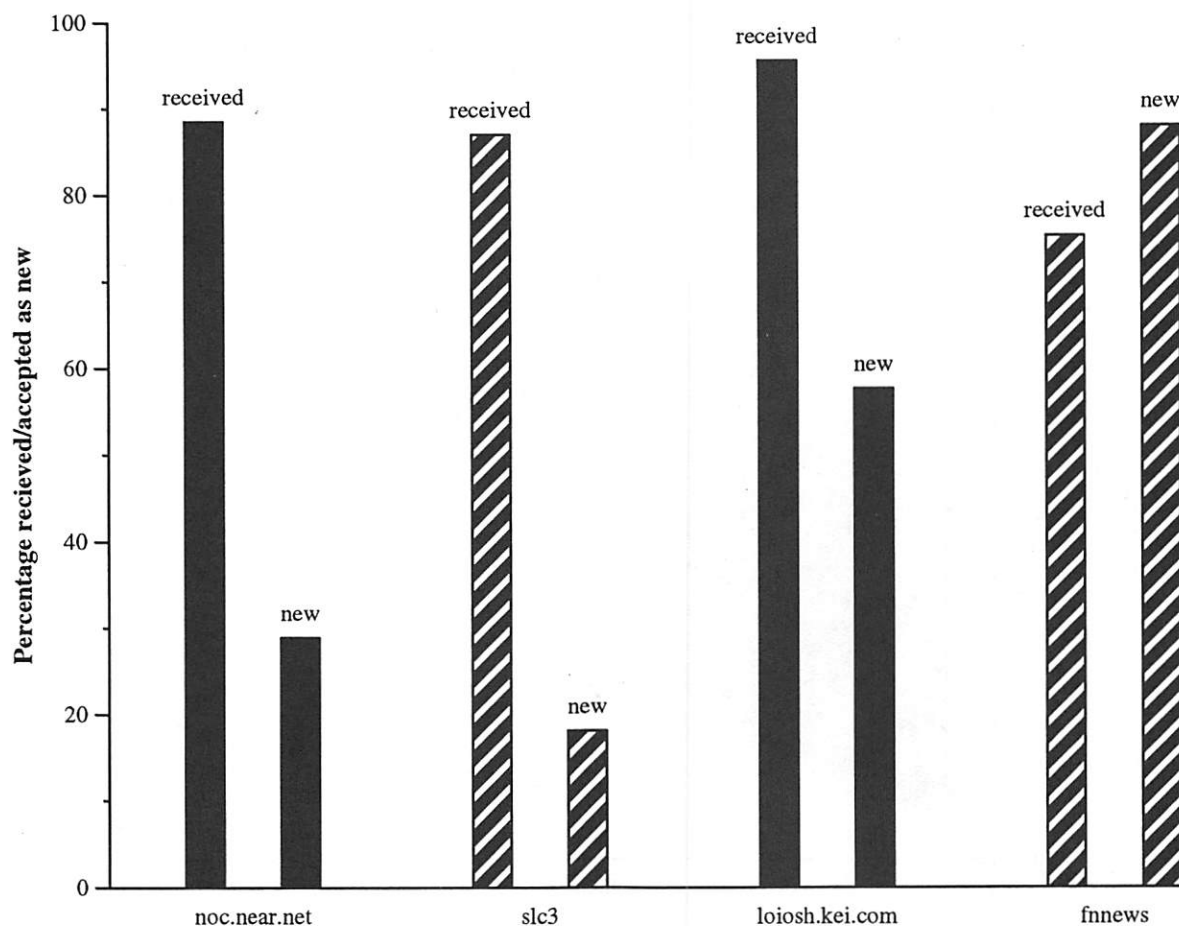


Figure 2

This bar chart [Figure 2] shows the percentage of complete articles (see Figure 1) that were received at the test sites. Additionally, it graphs the percentage of the received articles that were accepted by the NNTP server to which the listener offers the article. Because the “new” column is a percentage of received articles (as opposed to percentage of eligible articles) the “new” value can be larger then the “received” value, as is the case for the site fnnews.

The variation in the articles received at the various receivers can be mostly attributed to the widely varying placement of the receivers on the mbone. The closest is about two hops away, and the farthest is over ten.

The rather low percentages of “new” articles accepted at some sites is probably due to the quality of the “backup” news feeds to those sites. If the “backup” news feeds are sufficiently fast, they will deliver a given news article to the news systems before *Muse* can calculate the RSA signature and multicast the article.

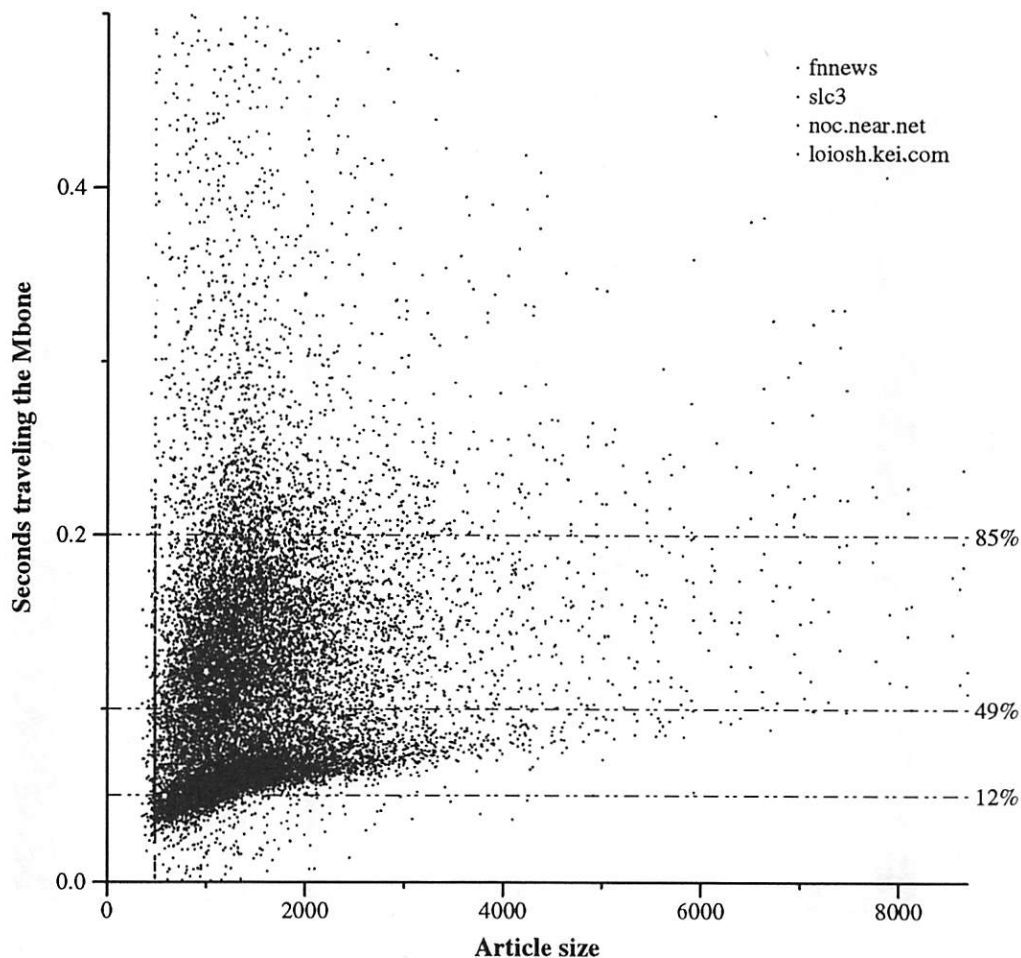


Figure 3

This scatter plot [Figure 3] shows the time in travel on the *Mbone* for several thousand articles. The major points of interest in this graph are the average small size of the articles and the low latency (almost always less than 200 milliseconds, and in fact quite often less than 100 milliseconds) time of the *Mbone* as a transport network. It is also interesting that the article size has a minimal effect on the time it takes to travel the *Mbone*.

There are several anomalies on this graph. A small number of articles (about five) appear to have arrived via before they were sent - we think this is most likely due to the clocks on the sending and receiving machines being out of phase. (Although all machines involved are synchronized using NTP, all the clocks involved will not match exactly in all cases. This is especially true for Sun SPARC machines running SunOS which disable clock interrupts when writing to the console.) There is also the smattering of times far above the mean. One possible cause of this is the fact that the timings are from user-space to user-space, and thus are affected by paging activity and competition for CPU time with other processes. Some of the delays are also attributable to the synchronous nature of the listener, which waits for any outstanding NNTP transaction to complete before it checks for new incoming news. (Both C News and INN can exhibit occasional relatively long delays in IHAVE transactions.)

(The rather noticeable "line" on the left side of the plot was caused by several hundred cancel messages sent out from brl.mil, closely clustered at and below size 487.)

8. Muse: Future Directions

- Faster Signing

One of the current limitations on the volume of news multicast is the fairly considerable cpu burden imposed by signing each individual message, which has caused us to limit ourselves to several hundred articles a day. We are looking at ways of improving this. (Either better code, a faster machine, or perhaps more than one article per packet.)

- Increased Multicast Selectivity

Once we can consider multicasting some large subset of all news traffic, it will become more interesting to break down the articles into multiple channels, to aid both receivers in selecting which articles to receive and the *Mbone* routers in insuring that each site receives only what it is interested in. This will be done by having the transmitter select a multicast group to send to based on the newsgroup of the article. Crossposts will probably be handled by broadcasting the article multiple times.

- Implementing full control message classes, message reply sending algorithms, timers, statistics gathering methods

Muse supports only the most basic control messages in this release of the software. Future work will be directed to more fully instrumenting the transmitter with a wider range of the possible control messages and the listener with the additional code to allow them to react to the control messages. Future control messages will allow for the gauging of the size of the listener pool at each distance from the transmitter, where the distance is measured in the hop-count to the transmitter. It is worthwhile to note that the transmitter of the control codes is not necessarily the machine that will handle the resulting replies from the *Muse* listeners. The basic control message will have a list of possible IP addresses to return the control message response to, to avoid overloading a single machine with the number of responses. Additionally, the response reply code will have a random-timer function in it, to avoid having all the machines that will be responding answering at once.

- A better receiver

The current receiver could be considerably improved in other ways as well. For example, the receiving and offering of the articles could be decoupled, which would allow the collection of slightly more accurate statistics. The receiver could also be able to offer a given article to more than one news server.

- What happens if the main transmitter or news sender goes down?

We are considering automatic switchover to another *Muse* transmitter or another news sender in the case of failure. A keep-alive control message would need to be implemented so that a secondary transmitter would not attempt to take over during periods with no news.

9. Conclusions

We developed *Muse* in order to make the propagation of USENET news more efficient, in both bandwidth consumed and speed of delivery. An important constraint on the solution was our requirement that it scale to hundreds or thousands of hosts. We believe that initial testing of *Muse* indicates that it can meet these goals. Specifically, utilizing the *Mbone* we can provide 200 millisecond simultaneous delivery for the vast majority of all articles to well-connected sites, with the number of possible receivers restrained only by the *Mbone* itself. We intend to pursue further testing and development of *Muse*, particularly in increasing the possible throughput.

10. Acknowledgments

- Thanks to Dave Hsu for suggesting the title for this paper. (hsu@pix.com)
- Thanks to Debbie Greenberg for proof-reading innumerable drafts of this paper. (dag@pix.com)
- Thanks to the authors of INN and C News for creating efficient news systems that inspired our own efforts.

- Thanks to Dennis Ferguson for the NTP timestamp code that we borrowed from xntp3.
- Thanks to those who have made the *Mbone* what it is today.
- Thanks to UUNET Technologies for allowing us to work and release this stuff, as well as running the *Muse* transmitter.

11. Availability

The source code for the client side implementation will be available via anonymous ftp from the host *ftp.uu.net* in the file */networking/news/muse/muse-client.tar.Z*.

12. References:

- [Adams87] Rick Adams, Mark Horton. *Standard for Interchange of USENET Messages* Request For Comments 1036, Falls Church, VA: Center for Seismic Studies, December 1987.
- [Deering89] Steve Deering. *Host Extensions for IP Multicasting* Request For Comments 1112, Stanford, CA: Computer Science Department, August 1989.
- [Fahn93] Paul Fahn. *Frequently Asked Questions About Today's Cryptography* Redwood City, CA: RSA Laboratories, September 1993.
- [Ioannidis91] J. Ioannidis, G. Maguire Jr. *The Coherent File Distribution Protocol* Request For Comments 1235, Columbia University: Department of Computer Science, June 1991.
- [Kaliski93] Burton S. Kaliski. *Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services* Request For Comments 1424, Redwood City, CA: RSA Laboratories, February 1993.
- [Kantor86] Brian Kantor, Phil Lapsley. *Network News Transfer Protocol: A Proposed Standard for the Stream-Based Transmission of News* Request For Comments 977, San Diego, CA: University of California, San Diego, February 1986.
- [Moore88] J. Moore. "Protocol Failures in Cryptosystems" *Proceedings of the IEEE*, Vol. 76, No. 5, Pg. 597, May 1988.
- [PKCS91] Public Key Crypto Systems #1. *RSA Encryption Standard, Version 1.4* Redwood City, CA: RSA Data Security, Inc., June 1991.
- [Rivest92] Ronald L. Rivest. *The MD5 Message-Digest Algorithm* Request For Comments 1321, Cambridge, MA: Massachusetts Institute of Technology, April 1992.
- [Robshaw93] M. Robshaw, Burton S. Kaliski. *On "Pseudocollisions" in the MD5 Message-Digest Algorithm* Redwood City, CA: RSA Laboratories, April 1993.
- [Salz92] Rich Salz. *InterNetNews: USENET transport for Internet sites* USENIX Proceedings, Summer 1992, Pg. 93. Boston, MA: Open Software Foundation, June 8-12, 1992.
- [Venema92] Wietse Zweitze Venema. *TCP Wrapper: Network Monitoring, Access Control, and Booby Traps*. Eindhoven University of Technology: Mathematics and Computing Science, July 1992.
- [Waitzman88] D. Waitzman, C. Partridge, S. Deering. *Distance Vector Multicast Routing Protocol* Request For Comments 1075, Cambridge, MA: BBN, November 1988.

13. Author Information

Kurt Lidl is a Senior AlterNet Engineer at UUNET Technologies. His current areas of concentration are dialup IP networking, *Mbone* deployment and process automation. Other interests include real-time combat simulation software, distributed computing and authentication systems, and public access Unix systems. Kurt attended the University of Maryland. He can be reached via E-Mail as *lidl@uunet.uu.net*.

Josh Osborne is a General Programmer at UUNET Technologies. His current areas of concentration are process automation and user services implementation. Other interests include real-time combat simulation

software, computer architecture design, kernel device drivers, WWW, fine beers, and computer languages. Prior job experience includes programming coin operated video games, and Unix system administration. Josh attended the University of Maryland. He can be reached via E-Mail as *stripes@uunet.uu.net*.

Joseph Malcolm is an AlterNet Engineer at UUNET Technologies. Interests include message systems, premises wiring plans, distributed environments, and WWW. He attended the University of Maryland. He can be reached via E-Mail as *jmalcolm@uunet.uu.net*.

The *refdbms* distributed bibliographic database system

Richard A. Golding[†]

Vrije Universiteit, Amsterdam, The Netherlands

Darrell D. E. Long[‡]

University of California, Santa Cruz

John Wilkes

Hewlett-Packard Laboratories, Palo Alto, California

Abstract

Refdbms is a database system for sharing bibliographic references among many users at sites on a wide-area network such as the Internet. This paper describes our experiences in building and using *refdbms* for the last two years. It summarizes the collection of facilities that *refdbms* provides, and gives detailed information on how well *refdbms* functions as a collaborative, wide-area, distributed information system.

1 Introduction

A bibliography in a paper serves two purposes: it acknowledges prior work, and it provides pointers to related efforts for readers. Useful bibliographies are accurate and complete, and hard work for a single person to generate. *Refdbms* is a system that assists this process. Bibliographies (like the one in this paper) can be generated easily and quickly, and their contents assured of high degrees of correctness.

Refdbms provides tools to add new entries, edit existing ones, search for interesting entries by keyword, and format the resulting bibliographies for L^AT_EX and Frame Maker documents. Each bibliographic entry is part of a reference database; any number of databases can be accessed together. Each database can be maintained by a disjoint group of people, but the combined databases are available for searching and retrieval. Several people, who may be physically dispersed, can access or update a single database.

Refdbms is not a single, monolithic system; rather, it consists of a set of tools that can be used to integrate it with other systems, such as text formatters and text retrieval tools. In this way the bibliographic databases can be used throughout the process of conducting scholarly research and writing.

In addition, *refdbms* databases can be replicated at many sites, providing reliable access to their contents even in the face of unreliable network connections, such as the world-wide Internet, or portable computers with only intermittent connectivity. Users at any of the sites can retrieve entries, submit new ones, or augment existing ones. All this is accomplished by exploiting a new class of weak-consistency protocols.

By comparison to other bibliography-maintenance tools, *refdbms* provides better search facilities, and roughly comparable output formatting facilities. Unlike them, however, its distributed, scalable nature means that several people can collaborate on the arduous task of generating – and keeping up to date – a bibliography pool of relevant and interesting items.

Refdbms is an example of a *collaborative* information system, by comparison to other wide-area information systems that use a *publishing* metaphor [Schwartz92]. It is also better integrated with the process of research and writing than most wide-area systems, and provides better availability and performance. At present, *refdbms* provides for either open, public collaboration, or private local use, though many other options are possible.

[†]Supported in part while at the University of California by a graduate fellowship from the Santa Cruz Operation.

[‡]Supported in part by the National Science Foundation under Grant NSF CCR-9111220 and by the Office of Naval Research under Grant N00014-92-J-1807.

The first versions of the *refdbms* system were written at Hewlett-Packard Laboratories [Wilkes91]. Over several years, this became the *refdbms version 1* system, which provided a bibliographic database shared within a single research group. In 1990, an effort began at U.C. Santa Cruz to build a portable version, as a first step toward using it as a prototype of a wide-area information system. A first version of this wide-area distributed system (known as *refdbms version 3*) was released in November 1992. Development since then has continued at U.C. Santa Cruz and the Vrije Universiteit Amsterdam to improve and extend the prototype.

Refdbms is publicly available software. It is currently in use at several sites, supporting at least 40 world-wide databases that contain nearly 14,000 computer-science-related bibliography entries.

The remainder of the paper is organized as follows. We begin with an overview of the *refdbms* functions, and compare these against related bibliography systems. In Section 3 we discuss the internal architecture of the system, how the pieces are connected, and how additional tools can be used with the system. We follow this in Section 4 by an analysis of the performance of the system in practice. In Section 5 we report some of the lessons we have learned and how we are using them to develop the next version of *refdbms*.

2 Services

Refdbms can generate bibliographies for documents from embedded citation information – for example, the bibliography for this paper was built using the system. Like Harrison and Munson [Harrison89], we believe that a good bibliographic database system should be integrated with the writing process, and should support other aspects of academic research. To this end, *refdbms* can also be used as a browsing tool to find interesting papers to read, it can be used for maintaining reading lists, and it has been used to manage technical report libraries.

Refdbms supports four general types of operations: storing, retrieving, sharing, and using references. Users can both retrieve information and update it, so that the effort of maintaining a database can be shared among them, and so they can evolve the database to meet their needs. Since we are concerned with geographically dispersed users, each database can be replicated at multiple sites, and updates made to one replica are propagated to other replicas.

In this section we briefly discuss these facilities. We start by defining references and databases, and then discuss how references are retrieved, modified, shared, and used.

2.1 References and databases

Refdbms maintains a collection of databases, each containing a set of references relevant to some topic. Each database has a name that is unique at the site. Table 1 lists some of the databases available. A site will usually maintain replicas of several databases, some of which will be publicly readable by every user on the site, and some of those databases will be shared with other sites as well. Users can also create their own databases, which are protected from other users to the degree of the permissions on the files storing the database. Each database must have a name that is unique within the site.

Refdbms stores references in a format reminiscent of that used by *refer* [Lesk78], with influence from BibTeX [Lamport85], as shown in Figure 1. A reference consists of a sequence of lines, each beginning with %<letter><space>. The letter indicates the *field* of which the line is a part. Some fields, such as the title (%T), consist of only a single line. Other fields, such as the list of authors (%A), consist of repeated lines. Finally, some fields, such as the extract (%x), are logically a single value spread over multiple lines.

Every reference has a *type*, such as TechReport or Article. This is indicated in the %z field, which must be the first line of the reference. Each reference also has a *tag*, such as Lamport78a, which is a unique name for the reference within its database. The tag is stored in the %K field, which must be the second line in the reference. Other fields, such as the title, authors, location, or date, may or may not be present depending on the type of the reference.

Refdbms allows *abbreviations* in a reference. An abbreviation is a sequence of letters and digits ending in a period, such as CACM. in the example in Figure 1. Most abbreviations are specific to a particular set of fields – for example, CACM. expands to Communications of the ACM only in a journal name (%J) field. Control files allow abbreviations to be expanded to a greater or lesser degree. Two control files are provided with *refdbms*, for full and minimal expansion, and it is easy to add a control file to conform to the particular conventions of a target journal or publisher.

The system enforces a few syntactic conventions on references. These include having the reference's type and tag as the first and second lines, and each type of reference is required to have some minimal set of fields (such as title, publisher, or author.) Some additional rules about formatting dates and names are enforced so every reference can be uniformly translated by simple programs.

TABLE 1: Some of the databases available in October, 1993. Several organizations have made available existing bibliographies used in their research.

Database	References	Contents
hpdbs	3412	General computer systems bibliography from HP Labs.
usenix	1337	Many Usenix Association publications through 1989.
parallel.miya	1274	Eugene Miya's parallel systems bibliography.
theory	1007	General references on the theory of computing.
inria.oos	965	Object-oriented operating systems bibliography from INRIA.
ucsc.grad.os	959	A canonical reading list for graduate OS courses; also contains references collected by students for term projects.
comp.doc.techreports	853	Technical reports announced on comp.doc.techreports.
logicprog	851	General bibliography on logic programming.
docbib	785	Database on document analysis and understanding.
dartmouth.io	234	References on parallel I/O systems, from Dave Kotz at Dartmouth.
inria.stochastic	188	Stochastic processes, from INRIA.
jacm	166	Selected articles from the Journal of the ACM.
x11bib	160	A collection of papers on the X Window System.
comp.os.research	148	Collected bibliographies from comp.os.research.
ieee.tcos	142	Contents of the IEEE TCOS (Technical Committee on Operating Systems) newsletter.
parallel	131	General bibliography on parallel computation.
compression	125	General bibliography on data compression.
ai	117	General bibliography on AI.
reinas	112	The REINAS project (UC Santa Cruz and MBARI).
inria.cache	104	Caching bibliography from INRIA.
inria.risc	99	RISC processor bibliography from INRIA.
crypt	90	General bibliography on cryptography.
comp.parle	77	Papers from PARLE '92 (Parallel Architectures and Languages Europe).
inria.bin-packing	77	General bibliography on bin packing, from INRIA.
compilers	75	General bibliography on compilers.
mitl.migration	53	References on process migration, from Fred Douglass at MITL.
ucsc.tr	53	Technical reports from UC Santa Cruz.
concurrent	48	General bibliography on programming current systems.
chorus	31	References on the Chorus operating system.
continmedia	27	General bibliography on continuous (multi-) media.
ucsb.cs.tr	15	Computer Science technical reports from the University of California at Santa Barbara.
hashing	14	General bibliography on hashing algorithms.
graphics	8	General references on computer graphics.
rfc	6	Selected Internet Requests for Comments.
Total	13743	

A reference tag is a unique name for the reference within a database. However, it is also supposed to be mnemonic and user-specified. Since two users could simultaneously try to add two different references with the same tag, the system internally uses a machine-generated unique identifier consisting of a host address and timestamp for each reference, and modifies the tag of one of the references to ensure its uniqueness. The internal unique identifier is invisible to the user. (The way the system copes with two simultaneous additions of the same reference is discussed in Section 2.3.)

When a reference must be unambiguously named among all databases, a *qualified* tag can be used, which has the form `<dbname>:<tag>`.

The *reference path* organizes a user's view of the databases available as a list of databases and the search order within them. The default is to construct a list by concatenating a file `.refpath` in the user's home directory with a list of the publicly available databases. This can be overridden by specifying the reference path explicitly on the command line, or in the `REFPATH` environment variable. Reference paths can nest: one element of a path can refer to a file containing additional path information. Further, user environment variables can be used within a path, so that complex context-dependent paths can be created when needed.


```

%z Article (the type)
%K Lamport78a (the tag)
%A Leslie Lamport
%T Time, clocks, and the ordering of events in a distributed system
%J CACM.
%V 21
%N 7
%D 1978
%P 558 565
%x The concept of one event happening before another in a distributed
%x system is examined, and is shown to define a partial ordering of
%x the events. A distributed algorithm is given for synchronizing a
%x system of logical clocks which can be used to totally order the
%x events. The use of the total ordering is illustrated with a method
%x for solving synchronization problems. The algorithm is then
%x specialized for synchronizing physical clocks, and a bound is
%x derived on how far out of synchrony the clocks can become.
%k causal consistency, asynchrony, happens before
%k clock synchronization

```

FIGURE 1: An example reference.

2.2 Retrieving references

Refdbms provides two ways of retrieving references: name-oriented and content-oriented. Name-oriented retrieval uses the reference tag as a unique name for a single reference. Content-oriented retrieval searches a database for references matching a query, returning the tags of those references that match. Retrieval by name is useful when a specific reference must be unambiguously named, while searching is more useful when the results are not so definitely known.

Searching is done by keyword. The system maintains an inverted index of words from the author, editor, title, and keyword fields. These words are *wordstemmed*, so that timing, times, and time all are indexed and searched as time.

For example, one can search the databases by keywords:

```

oak> refsearch Lamport time
comp.doc.techreports:abadi92
hpdb:lamport78a
hpdb:lamport84
hpdb:lamport86a
hpdb:lamport87
hpdb:lamport89b
ucsc.grad.os:lamport78
inria.oos:lamport78
inria.oos:lamport87

```

The text of one of these references can be retrieved using the `refget` command:

```

oak> refget hpdb:lamport78a
%z Article
%K hpdb:Lamport78a
%A Leslie Lamport
%T Time, clocks, and the ordering of events in a distributed system
%J CACM.
%V 21
%N 7
%D 1978
%P 558 565
(etc.)

```

The `refget` command can also expand the abbreviations in the reference.

In most circumstances one wants to see the text of the results of a search. The *reflook* command is a shorthand for piping the results of *refsearch* to *refget*.

Refdbms also provides *active* searches: a user can specify a query once, then periodically run the *refinterest* program to find what references have been added to local databases that match that query.

2.3 Modifying references

References can be added, changed, and deleted from a *refdbms* database. The programs that process add and change updates enforce the syntactic correctness constraints discussed earlier. The effects of an update are not immediately visible; instead, the update is written to a log that is processed periodically. An update must be received by every replica of the database before it can be processed. The time required depends on whether all the sites holding replicas are available; how often each site propagates updates; and how many databases there are.

Users at different sites can submit conflicting updates. These conflicts are resolved by holding an update until it has been received by every database replica, and by not processing an update until it can be globally ordered with respect to other potentially-concurrent updates. There are three sources of conflict:

1. Two references added with the same tag. This is resolved by ordering the add operations, and adding a suffix letter to the tag of the second. For example, if the tags of both references were *Lamport78*, the tag of the second reference would be modified to *Lamport78a*.
2. Two concurrent changes to the same reference. We use two techniques to resolve conflicts: first, we *avoid* many conflicts by transmitting the *changes* each update operation makes to the reference, rather than a complete image of the reference. The difference is done by field, and different fields have different rules: the title or extract is overwritten, while lines can be added to or removed from the list of locations where the paper can be obtained. Second, the differences are processed in a definite order by every database replica. This yields a consistent view of the reference when all updates have been processed – “single-copy” consistency.
3. Deletion and other operations. Changes to a deleted reference are discarded, as are attempts to delete it more than once.

Users may need to retrieve a new or modified reference while the database is waiting to propagate the update. For example, a user might add a reference to an article they have just read, then want to embed a citation to it in a document they are writing. Unfortunately, the tag of a new reference might have to be modified to make it unique when the update is finally processed, but the user must embed the tag into their document. We felt that immediate access was important enough that we had to allow it, but we were not willing to compromise the idea that the tag is a stable name for a particular reference. Our solution was to maintain two versions of a reference: a stable version and a “pending” version. A pending version is created whenever a reference is added or changed, and can be retrieved using a tag of the form *(dbname):(tag).pending*. When all outstanding updates for a reference have been processed, the pending version is deleted, so references embedded in user documents become invalid. This mechanism allows a user to explicitly state that they are willing to use information that may change, while allowing them to detect that the change has occurred. We considered another approach, where a user could qualify a tag by version or time information, but rejected it as too complex and unnecessary.

2.4 Sharing references

Many information systems have used the *publishing* metaphor: an individual or organization creates a database and makes it available for others to read. We are interested in a more flexible approach, where users can *collaborate* to manage a database. At the same time, *refdbms* has the flexibility to support publishing where it is appropriate.

Databases are the unit of sharing in *refdbms*. A user can create a new database and designate it either private or public. A private database is placed on that user’s personal list of databases. Other users can add it to their personal database list if the files in the database allow them read access. A public database is usually placed in a subdirectory of a public directory, and is readable and writable by all users at the site. Local operations on a database are coordinated by locking database files.

A public database can be made available for sharing with users at other sites by *exporting* it. Users at other sites can then *import* a replica of the database. An update made to any replica will be propagated to every other replica, as discussed in the last section. Updates can be restricted by exporting the database read-only to other sites, so that updates can only originate from the site that first exported the database – supporting the publishing model.

TABLE 2: Other bibliography-database systems.

System	Formatter	Typed	Distribution	Indexing	Search	Citation
refer	troff	no	no	nonincremental	absolute keyword	content
Tib	(La)T _E X	no	no	nonincremental	absolute keyword	content
BibLX	troff	no	remote query	nonincremental	absolute keyword	content
Scribe	Scribe	yes	no	no	no	name
BibT _E X	L ^A T _E X	yes	no	no	no	name
<i>Refdbms</i>	L ^A T _E X Frame Maker	yes	replication	incremental	word stem	name

2.5 Using references

We believe that it must be possible to integrate an information system with the work processes it supports – for *refdbms*, the processes of research and writing.

One can use *refdbms* to learn of interesting papers as their references are added to databases using an experimental notification facility, which allows a user to specify a set of keywords that they find interesting. As new references are added to databases, a filter is invoked and those references that match a user's keywords will be mailed to them. For example, a user can create a file `.refinterest` in their home directory:

```
query : distributed
query : operating system
```

then run the `refinterest` program to find all the references that match the keywords “distributed” or “operating system” that have been added in the last few days.

The support for L^AT_EX and Frame Maker documents uses BibT_EX. In the L^AT_EX source, one embeds the tag in a `\cite` command:

```
...that uses synchronized clocks \cite{hpdb:Lamport78a}.
```

One then builds a BibT_EX database from the references cited in the L^AT_EX source using the `refbibtex` utility, and processing proceeds as always. The `refmaker` command is the equivalent for Frame Maker documents.

2.6 Related systems

2.6.1 Other bibliographic database systems

Several other bibliographic database systems are in use, including *refer*, *Scribe*, *BibT_EX*, *Tib*, and *BibLX*. Table 2 summarizes these systems. In this section we will consider only those systems that integrally support the writing and research processes. None of the other systems listed support collaborative maintenance of a database.

The *refer* system supports the *troff* family of formatters [Kernighan78, Kernighan81], and was the original inspiration for *Tib* and *BibLX*. *Tib* [Alexander87], which supports T_EX, was derived from an earlier similar formatter called *bib*, which in turn was derived from *refer*. The reference file format is almost identical to *refer*, except that many field types have been added to support documents that have been translated from other languages, and the system comes with control files that describe many different styles of bibliography and citations. The *BibLX* system [Rodgers90] supports the *troff* formatter, and unlike other systems allows queries to remote databases.

The main functional differences between these systems and *refdbms* are that (1) the type of each reference is made explicit in *refdbms*, but left implicit in *refer*, *BibLX*, and *Tib*; and (2) *refer* reference entries don't have a unique tag by which they can be identified: instead, “sufficiently many” keywords have to be given to identify a single reference. We believe that an imprecise citation is inappropriate when large, changing databases are used because a citation that uniquely defines a particular reference one day might match many references the next. We also believe that type information helps to improve the accuracy of information in the database and improves the quality of translations into other forms.

Scribe was developed as a research vehicle to demonstrate the practicality of separating document content from form. A part of that demonstration was a nicely designed bibliography package. The *Scribe* program is now commercially available, at a not inconsiderable cost. *BibT_EX* [Lamport85] is an add-on package and program for the L^AT_EX formatting system that uses (essentially) the *Scribe* format for its references. Neither system provides support

TABLE 3: Other wide-area systems.

System	Model	Updates	Replicated	Active
Prospero	files	no	no	no
Alex	files	no	no	no
AFS	files	yes	yes	no
Jade	files	yes	no	no
Clearinghouse	name service	yes	yes	no
DNS	name service	no (publish)	yes	no
UNS	name service	no (publish)	yes	no
Archie	database	no	yes (not transparent)	no
Gopher	document graph	no (publish)	no	no
WAIS	full-text search	no (publish)	no	no
WWW	hypertext	no (publish)	no	no
Indie	general distributed index	no (publish)	yes	yes
Usenet	messages	yes (post)	yes (not transparent)	yes
<i>Refdbms</i>	bibliography	yes	yes (not transparent)	yes

for searching bibliography files: their purpose is to supply data for formatting, not to act as a database, though recently some tools have been made available to index and search BibTeX files. Further, although the format is elegant, it is not easy to use text-processing tools like `grep` and `awk` with it. *Refdbms* uses the type and tag from Scribe and BibTeX, but uses a basic reference format similar to `refer` for ease of manipulation by simple programs.

2.6.2 Other wide-area systems

Several information systems are available on the Internet. Some of these systems extend the file system model to span wide-area networks, some provide naming services, while others use an information-retrieval model. Table 3 summarizes several of these systems.

We have several criteria for judging a wide-area system. Availability and reliability are first among these: if the system cannot provide service, it is not useful. We believe that replication is an absolute requirement for a reliable wide-area system because it reduces the probability that the service is unavailable, decreases the mean network distance between client and server, and works to share processing and communication load across many machines. Some systems, including *refdbms*, provide non-transparent replication, where users must explicitly interact with one local replica, even though the overall service is replicated. Unreplicated systems fare even worse when one uses a disconnected computer. Unless the system can place a copy of interesting information on the local system, the service is completely unavailable as long as the computer is not connected. Cost is also important, including the cost in both time and money to set up a site, to begin sharing information with others, and to maintain the system. Finally, an active system can inform users of new information that is likely to be of interest to them.

The Prospero [Neuman92], Alex [Cate92], Jade [Rao93] and AFS [Howard88] file systems, among others, allow users at different sites to share different forms of information and to integrate this information with other tools. Unfortunately, they *only* provide the semantics of a file system: there is no possibility of supplying application-specific semantics to the information shared among different sites. These systems therefore cannot properly reconcile conflicting modifications made by different sites in all cases.

Name services, including the Xerox Clearinghouse [Oppen81], DNS [Mockapetris87], and Cambridge UNS [Ma92], implement specialized databases for translating names to addresses. All of these systems have very high reliability requirements, and so all provide replication. All restrict updates to part of the database to the organization that has authority for that part. The Clearinghouse and UNS systems use weak consistency replication techniques similar to those used in *refdbms*.

Other systems provide a read-only mechanism for *publishing* information. These systems include Archie [Emtage92], which allows users to search a database of files available for anonymous FTP; Gopher [Anklesaria93], a menu-oriented "document delivery system"; the WAIS [Kahle89] full-text search system; and the World Wide Web (WWW) [Berners-Lee92] global hypertext system. Except for Archie, these tools do not integrate with other tools.

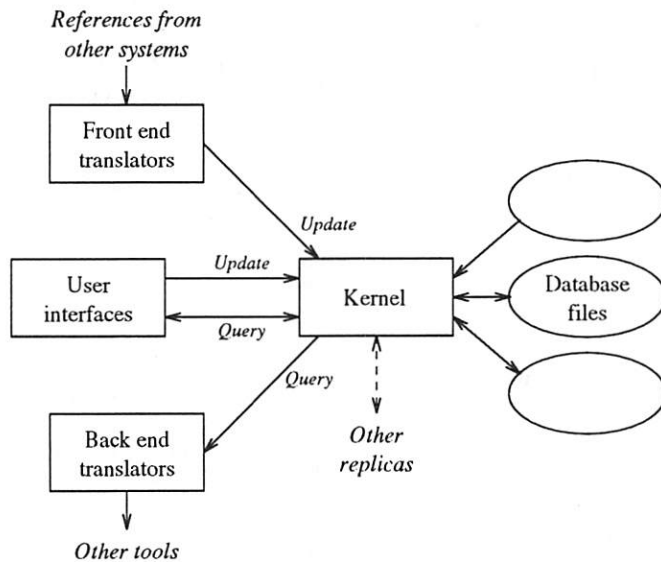


FIGURE 2: General architecture of the *refdbms* system.

Archie achieves better integration by using the Prospero file system protocol to connect clients to servers. Most of these systems have severe performance problems because of their lack of replication.

The Indie distributed indexing system [Danzig92] uses an interesting architecture that is an alternative to that used in *refdbms*. The Indie system consists of a number of distributed indexes, each of which collect “interesting” information from other information sources, and which can provide this information to other indexes. This is fundamentally a system for storing and discovering information published by other sources, and any changes must be made at the original source of the information.

Usenet is an often-overlooked information system, even though it is currently clearly the largest such system. Users can post messages to and read messages from a newsgroup.

3 Architecture

The *refdbms* system is divided into four parts: the kernel, which provides the database functions; front ends, which translate references from other formats; user interfaces, which allow the user to interact with the kernel; and back ends, which integrate *refdbms* with other tools. Figure 2 shows these components.

In this section we will discuss the implementation of each of these parts.

3.1 Kernel

The kernel implements the basic database functions, including updates, indexing, search and retrieval, and distributed operation. It is structured as a number of separate programs and libraries, and other parts of the system, such as user interfaces, use the kernel either by invoking the programs or calling library routines. Figure 3 shows the parts of the kernel.

There is one *submission* program for each of the kinds of updates that can be made to a database. When adding and changing a reference, the programs read a copy of how the reference should appear. The add program writes the new reference to the log in its entirety, while the change program computes a difference between the current value and the new, and stores that difference in the log. The deletion program merely takes the unique identifier or tag of the reference and writes a deletion record to the log.

The *log* stores the updates that have not yet propagated to every replica of the database.

The *replication* programs maintain the list of replicas for a database, propagate references between replicas, and control their creation, importation, exportation, and removal. Section 3.2 discusses this in detail.

Once a reference has propagated to every replica, it is *posted* to the database proper. In doing so, the reference is written to the references file for the database, which is indexed by the reference’s unique identifier. A tag index entry

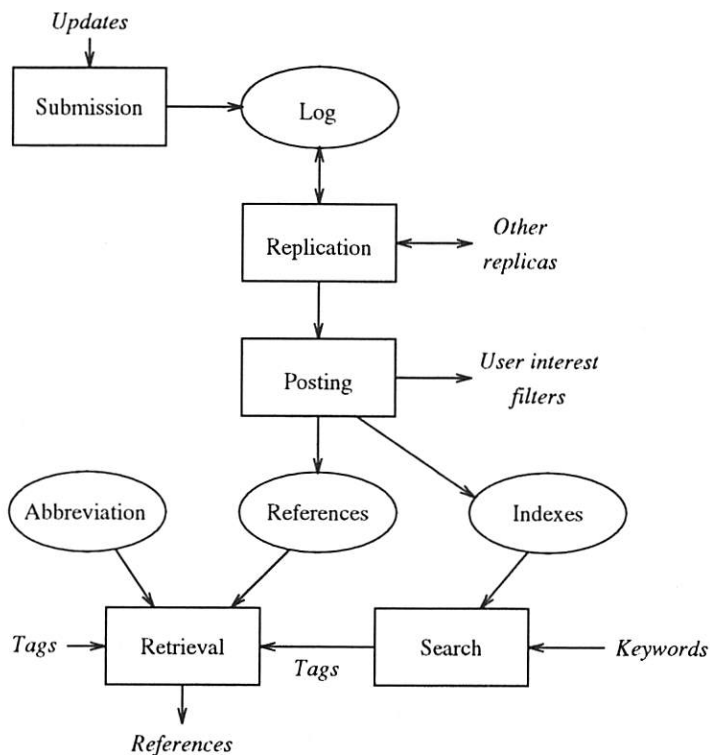


FIGURE 3: Components of the *refdbms* kernel.

is recorded to map the reference's tag to its unique identifier. In addition, the posting program incrementally updates an inverted index of the words in the references' keyword, author, and title fields.

The *search* program uses the inverted index to find the references that include some words. The words, both in the index and in the query, are wordstemmed: a *stem* or *base word* is computed for each word, and matching uses the stems. This ensures that trivial differences in a key, such as plurals, do not affect the search.

Finally, the *retrieval* program can retrieve references by tag. It includes a way to expand abbreviations.

We used several publicly-available libraries to build the *refdbms* kernel. These include Henry Spencer's *regexp* library for regular expressions; wordstemming routines from the *ispell* program maintained by Geoff Kuenning; and B-tree and hashed file management from the *db* library from the CSRG at U.C. Berkeley.

In addition, we have written a number of routines to assist in making the system portable, providing a functional interface behind which most system idiosyncrasies – such as omissions from the standard libraries, system logging facilities, and file locking – can be hidden.

3.2 Replication

Each exported *refdbms* database can be replicated at many sites. We used the *timestamped anti-entropy* (TSAE) weak consistency group communication protocol [Golding92b] to coordinate the replicas. We have described the overall architecture we are prototyping in *refdbms* elsewhere [Golding92a]. The current implementation only provides the essential parts of that architecture: database replication using a weak-consistency group communication protocol.

Weak consistency allows replicas to diverge for short periods of time, but guarantees that all of them will eventually receive every update and that they will compute the same results upon processing the updates. Divergence provides two important benefits: it allows the system to tolerate disconnected and crashed sites, and it makes the delay required for propagation invisible to the client update program. Divergence is acceptable in a system like *refdbms* as long as all the copies reach agreement reasonably quickly.

The TSAE protocol works as follows. When a client program submits an update, that update is written as a message in the log. A few times each hour, a cron job executes the *refae* program. If this program finds that there are updates that need to be propagated, it randomly selects another replica and opens a TCP connection to the *refdbmsd* daemon at that site. They perform an anti-entropy session, in which each of the two programs forwards to its partner any

update messages it has received and that the other has not. They also exchange information about the status of the group, including updates to the list of replicas and summaries of the messages each replica has received. After the session has been committed, the programs on either end run the `refpost` program in case some messages have now been acknowledged by every replica, and have become eligible for posting. The proofs that this protocol correctly propagates messages and analyses of its performance can be found elsewhere [Golding92b].

Another set of programs allow users to maintain the replicas at their site. `Refcreatedb` creates a new database, which can then be exported using `refexport`. If someone at another site wants to use a replica of the database, they use the `refimport` program, which contacts the daemon at some other site (the new site's "sponsor") to retrieve a complete copy of the database. When someone no longer wants to maintain a replica, they use the `refleave` program.

Each replica of a database has an associated *privilege*, which is one of `member`, `rwmember`, or `remember`. The latter two privilege levels allow some databases to be used for publishing information by not allowing updates except from the publisher. Users of a `member` replica can update the database, and any new replica sponsored by this replica will also be a full member. If a replica has `rwmember` privilege, its users can update the database, but any sponsored replica will be a `remember`. A `remember` copy only allows queries, not updates. In practice, the current mechanism can be subverted with moderate effort, and we are considering whether to provide more resilient authentication mechanisms to prevent this.

The system includes facilities for logging statistics on traffic and for checking other servers.

3.3 Front ends

The front ends translate references from other bibliographic systems into *refdbms* format. At the time of writing there are translators for BibTeX databases and refer databases.

The BibTeX translator uses `bibtex` itself to perform the translation. The `bibtex` program reads a list of citations, reads the references from a file, and parses them. It then interprets a special BibTeX style program that performs the actual translation.

`Refer` uses untyped references, so the translator must infer the type of a reference from its contents. Further, over the years different refer databases have evolved different, sometimes incompatible, conventions for entering references. The translator uses a set of rules to guide type inference and translation, and the standard rules can be augmented or replaced to adapt the translator to the peculiarities of each database. We have translated several thousand references using this tool, and found that with a few augmenting rules it is able to correctly translate a reference in the vast majority of cases.

3.4 Back ends

The back ends integrate *refdbms* with document processing tools. At the time of writing L^AT_EX and Frame Maker documents are supported.

Integration with both L^AT_EX and Frame Maker is currently done using BibTeX. For L^AT_EX, the back end scans a document, finding the `\cite` commands, retrieves the appropriate references from the databases, and builds a BibTeX file from them. It then uses the `bibtex` program to format the references. Frame Maker support is similar: a utility converts the document to text form, then scans it for citations. The citations are retrieved and formatted using BibTeX, converted to Maker Markup Language, and imported into the document.

3.5 User interfaces

The system includes three user interfaces: a command line interface, a GNU Emacs environment, and an experimental X application based on the XView toolkit.

The command line interface includes three programs: `refnew`, to enter a new reference; `refchange`, to edit a reference using a text editor; and `refdelete`, to delete a reference from the database.

The GNU Emacs interface is currently the most sophisticated interface for adding and changing references. It is an Emacs major mode that places each reference in its own buffer. The mode provides field-based manipulation (for example, tab to move to next field); justification; syntax checking; and submission. Three functions create new buffers: `ref-new` creates a new reference, inserting an empty template; `ref-retrieve` prompts for a tag and retrieves that reference; and `ref-copy` clones a buffer.

The experimental X application uses the XView toolkit, which provides an Open Look interface style. The application consists of a control panel from which other operations are invoked. The primary operations – querying and updating databases – can be initiated directly from the panel, while other operations are invoked from menus.

4 Performance

We have considered a number of performance measures in evaluating *refdbms*. These include the time required to perform basic manipulations, such as retrieving and searching for references; the time required to build indices; the storage required; and the network traffic produced.

Where appropriate we compare *refdbms* with *refer* and BibTeX. We used the *bib* system [Budd82] and the *Tib* system (version 2.1) for *refer*-format references. We used the *bibindex* program, version 2.2, for indexing BibTeX references.

All our performance measures were made on *frans.cs.vu.nl*, a diskless Sun SparcStation 1+ with 16Mbytes of memory running SunOS 4.1.2. The system was running X and a number of applications, though none of them were active during the measurements. This setup was selected as a typical workstation environment.

4.1 Retrieval

We characterize retrieval performance by the time required to retrieve some number of references, given their tag. We were unable to construct a fair comparison between *refdbms* and other systems, since *refer* does not use unique identifiers, and BibTeX does not index by identifier. We analyzed both the case where database files are not likely in memory, and when they have been cached. Under SunOS, this implies that it is likely that the virtual memory system has not yet gotten rid of pages holding file data.

A linear least-squares fit on these data suggest that the total time, in milliseconds, to retrieve r references from d databases, when not cached (t_n) and cached (t_c) is approximately

$$\begin{aligned}t_n &= 8r + 471d - 385 \text{ msec} \\t_c &= 4r + 103d + 87 \text{ msec}\end{aligned}$$

We caution that the data are insufficient to draw definite conclusions, other than that the time appears to be dominated by the time required to open a database, and that execution is several times faster if the data are already in memory.

4.2 Indexing

All three systems use an inverted index when searching for references by keywords. We compared the time required to build this index from scratch using *refdbms*; using the *invert* program distributed with *bib* and using the *tibdex* program distributed with the *Tib* system for *refer*-format references; and using the *bibindex* program, version 2.2, for BibTeX references. We report both the "real" (wall clock) time required and the estimated CPU time (in seconds):

System	Tool	CPU	Real
refer	invert	54.2	55
	tibdex	55.4	59
BibTeX	bibindex	20.8	24
<i>refdbms</i>	refbuildkeys	50.6	56

The *bibindex* program is significantly faster than the others. We believe that this is because that program indexes each field separately. We are encouraged that the simple *refbuildkeys* program – a shell script that uses several *sed* and *awk* scripts, and that in addition does wordstemming – is as fast as programs written entirely in C.

However, one rarely builds a *refdbms* index from scratch. Instead, the index is updated incrementally as the database is changed. When a new reference is added, its search keys are simply added to the index. When a reference is changed, all search keys for it are first deleted, then its keys are recomputed and added to the index. We investigated the time required to add, change, and delete various numbers of references from a database:

References	Add		Change		Delete	
	CPU	Real	CPU	Real	CPU	Real
1	3.4	6	8.6	19	5.7	12
2	3.4	6	8.5	15	6.0	11
4	3.6	7	8.8	17	5.9	11
8	4.1	10	9.2	19	5.9	11
16	4.5	10	9.8	20	5.9	12
32	5.2	11	10.4	21	5.9	11
64	6.3	12	11.4	22	5.7	11

4.3 Query

Most a user's direct interaction with a bibliographic database involves searching for keywords. We compared the time required to search for and retrieve references matching various keywords using *refer*, *BibTeX*, and *refdbms*:

Keywords	refer			BibTeX			refdbms		
	CPU	Real	Refs	CPU	Real	Refs	CPU	Real	Refs
Lamport	0.1	0	0	1.0	1	0	0.7	0	0
files	0.1	0	3	1.4	2	5	1.0	1	12
file	0.1	0	11	1.4	1	15	(same)		
adaptive routing	0.1	0	3	1.3	1	3	1.0	1	3
distributed systems		(failed)		2.1	2	131	1.5	1	92

The lookup program used in *refer* appears to perform all its operations in memory, causing it to be very fast. However, it is therefore unable to handle queries that return too many results. The *biblook* program can perform arbitrary boolean expressions, and can limit its searches to particular fields. This complexity makes it slower than the equivalent *refdbms* programs.

4.4 Storage

Refdbms uses a terse data format, similar to *refer*. We evaluated the storage required for one large sample database containing 1300 references. It appears that the basic syntax is as efficient as that of *refer* – the difference is the text of the reference type and tag – but a complete *refdbms* database requires nearly twice as much storage as does a simple *refer* database with its associated index. The same references translated into *BibTeX* format require significantly more space, but its inverted index is much smaller than that of *refdbms*, so the sum is smaller.

Metric	Refdbms		Refer		BibTeX	
	source	with index	source	with index	source	with index
size (bytes)	440553	1724416	413513	825331	686809	1054371
relative size	100%	100%	93.9%	47.9%	155.9%	61.1%

The bulk of the difference is in the hash file that stores the references themselves – it requires 1.2MB to store 440KB of text. Part of the overhead comes from the internal unique identifier: it averages 38 bytes, which is about 9% of the size of the data. This indicates that about 61% the hash file is overhead. Some of this is due to internal fragmentation in the hash file, and we are looking at ways to reduce the problem.

4.5 Traffic

Over the period of June 15 through October 27, 1993, there were 5078 updates propagated between replicas; of these, 4719 (92.9%) were new references; 147 (2.9%) were modifications; and 212 (4.2%) were deletions.

After an update has been submitted, a record of it must propagate to other sites before it can be applied. Figure 4 shows the distribution of the time between submission and posting to the database for updates to the *comp.doc.techreports* database between June 15 and October 27, 1993. The mean time to posting was 8942 seconds, or about two and a half hours. This database has had six replicas for most of that period, and other databases with as many replicas showed the same delay. Databases with one or two replicas required about ten minutes, on average, before an update could be posted, while one database, which had was subject to an extended network partition, had an average latency of five hours – mostly caused by a few updates being delayed for nearly a week.

5 Lessons learned

While the current version of *refdbms* shows that our architecture for wide-area systems is feasible, there are many improvements that we intend to make in future revisions.

The first is to decouple the identity of a replica from its location – in this case, the IP address of its host and TCP port number on which the daemon listens. This coupling prevents replicas from migrating between addresses, and restricts us to TCP as the transport protocol between replicas. In the next revision of the protocol, location and identification will be treated separately. A replica will be identified by an arbitrary unique value, and a list of $\langle \text{protocol, address} \rangle$ pairs will be maintained as the location of each replica. This will allow us to add new transports for communicating

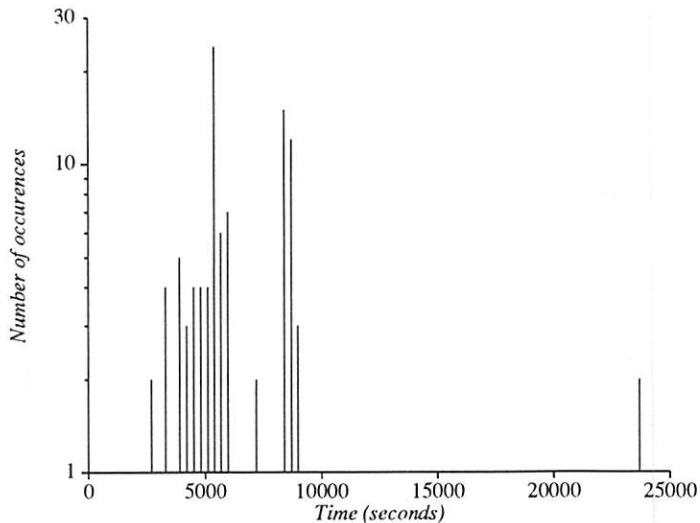


FIGURE 4: Histogram of times between submitting an update and it being applied.

between replicas, including UUCP, and it will allow the addition of new functions, such as a mechanism for remote queries on a replica.

Some databases contain several megabytes of information, and transferring this all in one go when a new database is imported can take quite a while over long-distance or low-bandwidth network connections. To assist this initialization, two new features will be added: first, the state transfer involved in creating a new replica will be split over multiple sessions, with each session transferring a well-defined fraction of the sponsor's state, until both the new replica and the sponsor have equal contents. Second, replicas containing only a subset of a database will be supported. The subset is defined by a set of predicates on references. In this way a site can reduce its storage requirement by only storing those references that are interesting, while forwarding queries it cannot satisfy to other replicas.

Formal analysis of the TSAE protocol identified an optimization that could be applied. As currently implemented, delivery of an update message is currently delayed until every replica has acknowledged receipt of that message. However, updates could correctly be delivered when the *local* replica acknowledges messages from any replica with lesser or equal timestamp, as long as all sites have approximately synchronized clocks. This allows an update message to be processed in $O(\log n)$ time after it is sent, where n is the number of replicas, rather than the $O(n)$ time currently required. The next protocol revision will include this improvement.

The prototype does not support fine-grained control over sharing. We are investigating how existing authentication and access control mechanisms can be used to support per-user protection, and how these mechanisms must be extended to do so.

We are investigating mechanisms for reducing the state information each replica must maintain. Currently, each replica must maintain information on every other replica. We are investigating a hierarchical scheme that would allow each replica to maintain information about $O(\log n)$ replicas. This approach would also make it easier for users to use an *optimistic* view of the database contents.

The system currently does not include any "name service" mechanism for maintaining information about the databases that are available. We are investigating this problem as part of research at the Vrije Universiteit on infrastructure for building wide-area applications.

As other researchers have found, writing a truly portable program of any considerable size is very hard. We have been able, for the most part, to encapsulate system differences in a small set of library routines. However, we have not had the same success in making a simple, portable mechanism for configuring, building, and installing the system. We are investigating several alternatives, including publicly available configuration packages, the X imake tool, and ways to construct a similar tool using the m4 macro processor.

We have also found that the simple database library we are using, which lacks crash recovery, is not adequate for constructing a reliable system: *refdbms* was initially developed on a set of systems that never experienced crashes, so this need was not experienced first hand by its developers. File locking has also been a problem, exercising what

appear to be bugs in the SunOS lock daemon. (These difficulties do not matter on more robust host systems, of course.) We are investigating mechanisms by which these problems may be circumvented.

6 Conclusions

A bibliography is an essential part of the process of writing and research. *Refdbms* is a tool for maintaining and using bibliographic information. It differs from other bibliography database systems and wide-area information systems in a number of ways:

- *Refdbms* emphasizes *collaborative* use of information by making it available for all its users to maintain, when appropriate, and allows information *publishing* when needed.
- It is *specific* to the research process, rather than a general-purpose tool.
- It is structured as a set of *tools* rather than as a single integrated system, so that it can become part of other, larger systems.
- It provides better *searching* capabilities than other bibliographic systems, including *active* searches that periodically inform the user of interesting new information.
- It is *distributed* and *replicated*, so that many users can share information while providing acceptable reliability and performance.
- It uses *weak consistency* so that it supports mobile computers and unreliable wide-area networks by strictly controlling communication.

The *refdbms* system is at the same time a prototype that allows us to evaluate our architecture for constructing wide-area applications. The results encourage us to believe that the architecture is suitable as a basis for many other applications, including name and authentication services, as well as other task-specific services. We are investigating some of these options.

The system is publicly available. The current version is available for anonymous FTP on `ftp.cse.ucsc.edu`, in the directory `/pub/refdbms`. A list of available databases can be found using `finger refdbms@refdbms.cse.ucsc.edu`.

Acknowledgments

The many *refdbms* users have contributed substantially to the system, both in code, bug fixes, and experience. At Hewlett-Packard Laboratories, David Jacobson wrote the original Emacs mode, and Carl Staelin first integrated the wordstemming code. The Mammoth Project at U.C. Berkeley under National Science Foundation infrastructure grant CDA-8722788 provided resources for testing early versions of the system. George Neville-Neil, Tage Stabell-Kulø, Peter Bosch, and Henri Bal all made helpful comments on this paper.

References

- [Alexander87] J. C. Alexander. *Tib: a TeX bibliographic preprocessor* (1987). Department of Mathematics, University of Maryland. Version 2.1.
- [Anklesaria93] F. Anklesaria, M. McCahill, P. Lindner, D. Johnson, D. Torrey, and B. Alberti. The Internet Gopher protocol (a distributed document search and retrieval protocol). Request for Comments 1436 (March 1993). Internet Engineering Task Force.
- [Berners-Lee92] Tim Berners-Lee, Robert Cailliau, Jean-François Groff, and Bernd Pollermann. World-Wide Web: the information universe. *Electronic Networking: Research, Applications, and Policy*, 1(2) (Spring 1992).
- [Budd82] Timothy A. Budd and Gary M. Levin. A Unix bibliographic database facility. Technical report 82-1 (1982). University of Arizona.

- [Cate92] Vincent Cate. Alex – a global filesystem. *Proceedings of the File Systems Workshop*, pages 1–11 (May 1992). Usenix Association.
- [Danzig92] Peter B. Danzig, Shih-Hao Li, and Katia Obraczka. Distributed indexing of autonomous Internet services. *Computing Systems*, 5(4):433–59 (Fall 1992). Usenix Association.
- [Emtage92] Alan Emtage and Peter Deutsch. Archie – an electronic directory service for the Internet. *Proceedings of the Winter Conference* (San Francisco), pages 93–110 (January 1992). Usenix Association.
- [Golding92a] Richard Golding. A weak-consistency architecture for distributed information services. *Computing Systems*, 5(4):379–405 (Fall 1992). Usenix Association.
- [Golding92b] Richard A. Golding. *Weak-consistency group communication and membership*. PhD thesis, published as Technical report UCSC–CRL–92–52 (December 1992). Computer and Information Sciences Board, University of California at Santa Cruz.
- [Harrison89] Michael A. Harrison and Ethan V. Munson. On integrated bibliography processing. *Electronic Publishing*, 2(4):193–209 (December 1989).
- [Howard88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81 (February 1988).
- [Kahle89] Brewster Kahle. Wide area information server concepts. Technical report TMC–202 (3 November 1989). Thinking Machines Corporation.
- [Kernighan78] Brian W. Kernighan and Lorinda L. Cherry. A system for typesetting mathematics. *Communications of the ACM*, 18(3):151–6 (March 1978).
- [Kernighan81] Brian W. Kernighan. A typesetter-independent TROFF. Computing Science technical report 97 (1981). Bell Laboratories, Murray Hill, NJ.
- [Lamport85] Leslie Lamport. *L^AT_EX: a document preparation system* (1985). Addison-Wesley Publishing Company, Reading, MA.
- [Lesk78] M. E. Lesk. Some applications of inverted indexes on the UNIX system. Computing Science technical report 69 (June 1978). Bell Laboratories, Murray Hill, NJ.
- [Ma92] C. Ma. *Designing a universal name service*. PhD thesis, published as Technical report 270 (1992). University of Cambridge Computer Laboratory.
- [Mockapetris87] P. Mockapetris. Domain names – concepts and facilities. Request for comments 1034 (November 1987). ARPA Network Working Group.
- [Neuman92] B. Clifford Neuman. The Prospero file system: a global file system based on the virtual system model. *Computing Systems*, 5(4):407–32 (Fall 1992). USENIX Association.
- [Oppen81] D. C. Oppen and Y. K. Dahl. The Clearinghouse: a decentralized agent for locating named objects in a distributed environment. Technical report OPD–T8103 (1981). Xerox Office Products Division, Palo Alto, California.
- [Rao93] Herman C. Rao and Larry L. Peterson. Accessing files in an internet: the Jade file system. *IEEE Transactions on Software Engineering*, 19(6) (June 1993).
- [Rodgers90] R. P. C. Rodgers and Conrad Huang. *BibIX 2.1 Manual* (January 1990). University of California Office of Technology Licensing, Berkeley, California.
- [Schwartz92] Michael F. Schwartz, Alan Emtage, Brewster Kahle, and B. Clifford Neuman. A comparison of Internet resource discovery approaches. *Computing Systems*, 5(4):461–93 (Fall 1992).
- [Wilkes91] John Wilkes. The refdbms bibliography database user guide and reference manual. Technical report HPL–CSP–91–11 (20 May 1991). Hewlett-Packard Laboratories, Palo Alto, California.

Biographies

Richard A. Golding received his B.S. degree in Computer Science from Western Washington University in 1987. He received his M.S. degree in 1991 and his Ph.D. degree in 1992, both in Computer and Information Sciences from the University of California, Santa Cruz. He is currently a researcher at the Vrije Universiteit Amsterdam. His research interests include distributed operating systems, wide-area systems, and object-based programming environments.

Darrell D. E. Long received his B.S. degree in Computer Science from San Diego State University in 1984. He received his M.S. degree in 1986 and his Ph.D. degree in 1988, both in Computer Science from the University of California, San Diego. He is currently Assistant Professor of Computer & Information Sciences at the University of California, Santa Cruz. He has published articles on protocols for data replication, host reliability, performance guarantees and high-speed I/O systems. His current research interests include distributed computing systems, high-speed I/O systems, performance evaluation and real-time management of large scientific data sets. Dr. Long is a member of the IEEE Computer Society, the Association for Computing Machinery and Sigma Xi. He is the chair of the IEEE Technical Committee on Operating Systems and Application Environments. He also serves on the editorial board of the *International Journal in Computer Simulation*.

John Wilkes graduated with degrees in Physics (B.A. 1978, M.A. 1980), and a Diploma (1979) and Ph.D. (1984) in Computer Science from the University of Cambridge. He has worked since 1982 as a project manager, and researcher at Hewlett-Packard Laboratories. His current research interests are in highly available high-performance storage systems, and in resource management in scalable computers. He also enjoys dabbling in photography, interconnects for scalable systems, and performance modelling. He particularly enjoys working with university colleagues on projects such as this one.

Filesystem Daemons as a Unifying Mechanism for Network Information Access

Steve Summit
Consultant, Seattle, Washington

Abstract

As the Net burgeons, new tools and protocols are being introduced to permit some orderly use to be made of the wealth of information available. These new protocols, however, often presuppose the use of new, nonstandard, highly interactive user interfaces. This paper presents a mechanism for unifying access to diverse network services through filesystem daemons, which allow network information services to be treated as if they were conventional files and directories, residing in the local namespace, and accessed transparently with standard tools. Besides normal filesystem operations (*open*, *read*, *write*, etc.), the daemons may introduce extended operations, which provide generic access to such features as network database lookup operations.

1. Introduction

As the Internet grows, and as more and more information becomes available, it becomes more difficult to locate information of interest. Numerous protocols, such as Gopher [Anklesaria 93], the World-Wide Web [WWW 93], and WAIS [Kahle 89, Davis 90] have been introduced in an attempt to make network browsing and information retrieval more convenient and productive [Krol 92]. These protocols, and their associated user interface tools, represent important advances, and they make the resource discovery problem much more tractable. However, since they are intended as browsing tools, their default user interfaces tend to be highly interactive.¹ Furthermore, as the new protocols are still somewhat experimental, their user interfaces are rather idiosyncratic. Though perfectly adequate for casual browsing, they do not lend themselves immediately to integration into larger toolsets.

It is natural to wonder whether that venerable old UNIX® notion that “everything’s a file” can be extended to these newer network information sources. (Conventional network files have of course been so available for some time; see section 9.) Under this extended paradigm, the filesystem interface becomes a *rendezvous point* between network services and local utilities. It is possible to explore gopherspace, or the World-Wide Web, or other information spaces, using one’s own preferred shell, moving around with *cd*, viewing and searching “files” with standard tools such as *more* and *grep*, copying them to one’s home directory with *cp*, and otherwise making use of a large, open, familiar, rich, programmable environment.

This paper describes a prototype implementation which provides such transparent access. The implementation is built up in several stages, and that structure will be reflected in the body of the paper. Section 2 introduces the idea of filesystem daemons, which permit a process to be invoked to handle any or all of the I/O associated with any name in the filesystem. Section 3 reviews the straightforward application of such daemons in implementing transparent remote filesystem access (e.g. via anonymous ftp). Section 4 extends the remote access paradigm to the realm of non-filesystem-oriented data such as gopher and WWW,

1. Even ftp, which the new protocols are intended to supersede, has this problem.

and proposes relaxing the distinction between files and directories (it is possible to view some pieces of structured or indexed information as both). Section 5 suggests extensions to the basic set of I/O operations, in order to permit advantageous use of such services as remote database search facilities. Section 6 describes details of the implementation, and section 7 discusses performance. Section 8 outlines open issues and future work; section 9 compares related work.

The implementation described runs entirely in user mode; echoing a refrain of USENIX contributions past, "no kernel changes are required."

2. Filesystem Daemons

The underlying mechanism on which the ideas in this paper are built is an extended filesystem tentatively named the Object-Oriented Filesystem, or OOFS². The salient feature of this filesystem is that any file may enjoy customized I/O handling via the services of an associated process, or *filesystem daemon*. In effect, each file may be treated as an object to which I/O messages are passed. (A similar scheme is described in [Bershad 88].)

Under OOFS, each UNIX system call which accepts a pathname is intercepted and the pathname inspected for the presence of a daemon attached to the referenced file, or to one of its parent directories. If a daemon is found, it is invoked, and a communication path is set up along which are passed messages which request further lookup and I/O operations. OOFS also intercepts system calls which accept file descriptors; I/O on descriptors which refer to daemon-opened files again results in messages being passed to the daemon for processing, rather than direct invocation of the conventional kernel routine. Daemons may respond to I/O requests in arbitrary ways, constructing the file's data on the fly, if they so wish.³

The protocol between a daemon and its OOFS-aware client is simple and—very importantly—extensible. (Further rationale behind, and ramifications of, this extensibility are discussed in section 5; details of the daemon interface protocol can be found in section 6.) For the purposes of the following discussion, it suffices to know that daemon requests are specified by text strings; most are named after the UNIX system calls they implement.

The OOFS mechanisms support conventional, slash-separated, UNIX-style pathname syntax. The semantics can however be arbitrarily tangled; as we shall see, various path components may end up representing access methods, machine names, etc. Furthermore, if a daemon requires any special parameters, these must typically be passed to it using daemon-specific syntax grafted on to the pathname. (Similar gyrations are discussed in [Roome 92].)

Filesystem daemons are obviously useful for purposes other than networking. They can be used to implement access control lists, self-decompressing files, self-extracting archives, versioning systems [Korn 90, Roome 92], mailbox directories, and NNTP-based `/usr/spool/news` directories, among other things, but those applications are outside the scope of this paper.

3. Remote Filesystem Access

Once things are set up so that daemons can intercept and provide special processing for certain pathnames, the groundwork is obviously laid for transparent network file access. For example, I have implemented a sort of "poor man's NFS" by attaching a daemon to the pseudodirectory `/ftp`. (This technique is very similar to an approach used by the Alex system [Cate 92].) This daemon handles all pathname components beneath the `/ftp` attachment point, such that pathnames of the form

`/ftp/machine/path`

are interpreted as a path accessed via anonymous ftp on a certain machine. The daemon acts as an ftp client, connecting to the specified machine's ftp server and performing appropriate ftp protocol transactions as

2. An admittedly obvious name.

3. Eggert and Parker discuss such "intensionalized" files extensively in [Eggert 93].

necessary to satisfy *its* client's (that is, the daemon's clients) I/O requests. Under this scheme, the `ftp` user command becomes superfluous; files can be copied by anonymous ftp with the `cp` command (or *moved* with `mv`). For example, the command

```
cp /ftp/ftp.nisc.sri.com/rfc/rfc1149.txt .
```

retrieves a file via anonymous ftp from site `ftp.nisc.sri.com`; no manual interaction with an ftp client is required.

4. Remote Non-Filesystem Access

The preceding two sections are by way of prelude; they do not represent anything that has not been done several times before. This paper's principal contribution is the idea of accessing, transparently and as if part of the local filesystem, non-filesystem-structured network information services.

4.1. Gopher

The Internet Gopher protocol [Anklesaria 93] presents a hierarchical tree of information nodes, residing potentially on many machines. Nodes are typed: some are simple pieces of text, others are "menus" (analogous to directories) pointing at other nodes, still others are simple search engines. A simple, connectionless protocol allows a node's contents to be fetched; nodes which are directories return information about each subnode contained: its type, the machine on which it resides, and the identifying tag by which it can be fetched.

The conventional user interfaces to the Internet Gopher are menu based: each time a selection is made from a gopher menu, the user is presented either with another menu, or with a piece of text or other information. The invocation style is highly interactive; if there is a way to save oneself a copy of an interesting node, it is neither the way that one saves interesting mail messages nor news articles nor files discovered while exploring the net using `ftp` or some tool other than gopher.

Gopherspace can be presented as a filesystem, however, by an OOFS daemon which maps gopher menu nodes to directories and other nodes to files. This daemon interprets pathnames of the form

```
/gopher/machine-name
```

as the root of the gopherspace tree on the host named `machine-name`, and it is therefore possible to explore gopherspace using one's favorite shell, using `ls` to see the contents of a "menu," `cd` to move among menus, and `cat`, `more`, or `cp` to view or save nodes of interest.

Figure 1 (on the next page) shows a sample interaction using the OOFS gopher daemon.

It must not be suggested here that the mapping from gopherspace to a simulated filesystem is perfect. The entries in gopher menus tend to be multiword titles, not the shorter, single-word names one expects to see in directories.⁴ As the example in Figure 1 shows, gopher nodes have neither meaningful owners, sizes, nor modification times.⁵

The mapping is not without compensating merits, however: to a user who prefers a conventional shell, and dislikes using different interfaces in different contexts, the aberrations visible when trying to treat gopherspace as a filesystem are no more jarring than the differences between a conventional gopher client and that preferred shell. More importantly, access to gopherspace via the filesystem and a programmable shell leaves open the possibility of building new interaction or processing utilities using a toolkit approach.

4. To ease the resulting burden slightly, the prototype gopher OOFS daemon implements a simple implicit wildcard scheme when matching pathnames against gopher menus: a name not otherwise matched will match a name of which it is an initial prefix, if unique. (Purists will note that this twist is entirely unnecessary, users could otherwise make frequent use of shell globbing when `cd`'ing through gopherspace.)

5. Some nodes, such as gopher search engines, may not even map to a proper UNIX "file type;" as the example shows, the `ls` command is unable to interpret the `S_IFMT` bits, and displays an unusual leading '?.'

```

$ cd /gopher/gopher.micro.umn.edu
$ ls -l
dr--r--r-- 1 0 0 Dec 31 1969 Information About Gopher
dr--r--r-- 1 0 0 Dec 31 1969 Computer Information
dr--r--r-- 1 0 0 Dec 31 1969 Discussion Groups
dr--r--r-- 1 0 0 Dec 31 1969 Fun & Games
...

$ cd Inf*
$ pwd
/gopher/gopher.micro.umn.edu/Information About Gopher
$ ls -l
-r--r--r-- 1 0 0 Dec 31 1969 About Gopher
?r--r--r-- 1 0 0 Dec 31 1969 Search Gopher News
dr--r--r-- 1 0 0 Dec 31 1969 Gopher News Archive
dr--r--r-- 1 0 0 Dec 31 1969 comp.infosystems.gopher
...

$ cat Abou*
This is the University of Minnesota Computer & Information Services
Gopher Consultant service.
...

```

Figure 1
Sample Gopher Session

4.2. World-Wide Web

The World-Wide Web [WWW93] is based on two functionally distinct ideas. The first is a connectionless information retrieval protocol, much like gopher's, by which a named text entity is retrieved from a server. The second, and more significant aspect of WWW is its hypertext model: in general, all text entities contain hypertext links to related entities. Interaction with WWW consists entirely of chasing links to narrow in upon information of interest (or just to explore).

The existing WWW interaction tools, even the lowest-common-denominator text-only version, are excellently written. Besides formatting WWW's embedded HTML (Hypertext Markup Language) constructs appropriately, they make it very easy to chase links of interest and wander around in the Web. It is not my intent to criticize the WWW interfaces; they are superior. But they are not the shell, and one cannot transparently use arbitrary UNIX commands from within them.⁶

Once again, however, a suitable daemon allows the Web to be accessed using familiar tools. The OOFS WWW daemon presents WWW text entities as conventional files which can be opened and read by any standard tool. At the same time, each text may also function as a directory: its subfiles or subdirectories (that is, the items it contains) are simply the text entities pointed to by its embedded links.

The OOFS WWW daemon, then, provides an example of the utility of relaxing the file/directory distinction. Each text entity is simultaneously a file *and* a directory. A program such as `cat`, which opens an entity conventionally, reads text, while a program such as `ls`, which opens it as a directory, reads directory entries.

6. To be sure, the WWW interfaces do provide shell escapes, and mechanisms for piping node text to arbitrary commands.

When exploring the Web using a shell and the OOFs WWW daemon, traversing a link to a new node is done with `cd`, and viewing the current node can be done with the formerly meaningless invocation

```
cat .
```

Returning to a previously-visited node is of course accomplished with

```
cd ..
```

Figure 2 shows a sample foray into the Web using a conventional shell and the OOFs WWW daemon. This example shows another issue of interest when presenting WWW text entities as files: the text contains embedded HTML formatting requests, which should usually be processed before display. It is an intriguing question whether an OOFs daemon should implicitly perform such formatting; for now, it does not, and in these examples the existing WWW line-mode browsing tool, `www`, is used *as a filter only* (indicated by its `-` option) to format the HTML for display.

```
$ cd /www/info.cern.ch/default.html
$ cat .
<HEAD><title>Overview of the Web</title></HEAD>
<BODY>
<h1>General Overview</h1>
There is no "top" to the World-Wide Web.
You can look at it from many points of view.
Here are some places to start:
<DL>
<DT><a href=http://info.cern.ch./hypertext/DataSources/bySubject/Overview.html>The Virtual Library</a>
<DD>A classification by information by subject.
...

$ cat . | www -
Overview of the Web

GENERAL OVERVIEW

There is no "top" to the World-Wide Web. You can look at it from
many points of view. Here are some places to start:
The Virtual Library[1]
A classification by information by subject.
...

$ cd 1
$ cat . | www -
The World-Wide Web Virtual Library: Subject Catalogue
THE WWW VIRTUAL LIBRARY

This is a distributed subject catalogue. See also arrangement by
service type[1] ., and other subject catalogues[2] .
...
```

Figure 2
Sample WWW Session

Accessing the Web using only a shell, `cd`, and `cat` may seem to mock the more sophisticated text formatting and linking features which the Web provides, and definitely skirts the edge of the mappings which are meaningfully appropriate under the "everything's a file" model. However, the ability to use a familiar,

general-purpose shell and toolkit at least partially compensates for the lack of full hypertext awareness. Furthermore, it is intriguing to contemplate the possibility of a general-purpose "hypertext shell" which could be used to explore both the World-Wide Web and other hypertext systems. It would be easy to write such a shell if the details of various hypertext schemes were hidden behind a common, filesystem-like interface; in fact, a few simple shell aliases can go a long way towards providing hypertext-like features within a conventional, but OOFS-aware, shell.

5. Extended I/O Operations

One of the more important services provided by network information retrieval protocols is searching or lookup. No matter how partial one is to one's own favorite `grep` variant or other tools, it is not practical to pull many megabytes of data over the network only to selectively discard most of it. A search or lookup operation provided by a network protocol allows the searching to be done on the machine which has local access both to the data and to any precomputed indices or inverted files.

The existence of these specialized search and lookup operations is a more compelling force which tends to lock users into the idiosyncratic user interfaces which support and are supported by the protocol. One's own existing tools are inevitably based on simple reads of chunks of data, and no amount of intelligent operation mapping by a daemon sitting at the level of a conventional I/O interface is going to be able to make use of a predefined search or lookup operation.

It is for this reason that the OOFS daemon protocol has been left very open and extensible. Both gopher and WWW, for example, provide simple keyword searches (as does WAIS; keyword searching is its whole purpose). It is difficult to see how a filesystem daemon can make use of these features: no existing I/O call can meaningfully be mapped to a search or lookup operation; no existing general-purpose tools presuppose the existence of such a facility. If, however, the filesystem is to be the rendezvous point between applications and information servers, we may contemplate the invention of a new, filesystem-level call which will map to these search and lookup operations.

In gopher and WWW, a search is performed against an existing object and yields a menu (under gopher) or a text entity full of links (under WWW) pointing at the objects which were found. Therefore, for gopher and WWW, we may devise a lookup operation which functions rather like `mkdir`: this new operation takes as parameters the name of an existing object, a search pattern, and a new name. If the search succeeds, a new directory entry, with the specified new name, is created in the searched-upon object, and points at the result of the search (which is actually a directory of search results).

This new operation, called *mdlookup* (for "make-directory lookup") is implemented by both the gopher and WWW daemons. A simple program, `lookup`, provides a shell-invokable interface to the new operation; `lookup` can be used to perform searches either in gopherspace or the Web. (A simple shell alias could encapsulate the `lookup` invocation, new name selection, and `cd` into the created directory, if successful.)

Figure 3 (on the next page) shows an example of the `lookup` command being used along with the WWW daemon.

The role of these extended operations—such as *mdlookup*—must be carefully understood. As they are neither part of the standard UNIX I/O interface nor utilized by standard tools, they may seem to be as idiosyncratic as the special-purpose user interfaces which they are attempting to replace. Their advantage is that they sit at the level of, and augment, an existing interface (namely the filesystem). They can therefore be used to build upon and extend existing, more standard operations, permitting efficiency and synergy without having to discard existing interfaces and toolsets.

```

$ cd /www/info.cern.ch/default.html/1/9
$ cat . | www -
                                The World-Wide Web Virtual Library: Computing
                                COMPUTING
                                Information categorised by subject. See also other subjects[1] .
...
Jargon[7]                        Computer hacker's jargon index
...

$ cd 7
$ cat . | www -
                                Collection 'Hacker's Jargon'
                                HACKER'S JARGON

                                A[1]
                                B[2]
                                C[3]
...

$ lookup . kluge search1
$ cd search1
$ cat . | www -
                                jargon?kluge
THE FOLLOWING OBJECTS MATCH 'KLUGE' IN COLLECTION 'HACKER'S JARGON'
    kluge[1]
    kluge around[2]
    kluge up[3]

$ cd 1
$ cat . | www -
                                kluge
                                KLUGE
kluge: /klooj/ [from the German 'klug', clever] 1. n. A Rube
Goldberg (or Heath Robinson) device, whether in hardware or
software.
...

```

Figure 3
Sample lookup Operation

6. Implementation Details

The decision to go with a user-mode implementation was made for several reasons. One was pragmatic; machines with kernel sources and tolerant users were not available. Secondly, user mode code can be markedly easier to develop and debug than kernel code [Warnock 84]. Finally, there is an undeniable challenge in implementing things in user mode which classically "belong" in the kernel. The choice is not without its disadvantages, of course: it is somewhat difficult to preserve fork and exec semantics of open files, and unless a system supports dynamic linking or run-time interception of library routines and system calls (as many modern systems in fact do), it can be a nuisance to have to relink large numbers of programs.

Since this is a user-mode implementation, eschewing kernel modifications, it does not rely on any modifications to the on-disk filesystem structure. Instead, the presence of a daemon attached to any file is recorded in a hidden file in the same directory. A central "fallback" daemon attachment file may also be

used; this file allows users to attach daemons to files or directories (e.g. \$MAIL or /) for which the parent directories are not writable.

The implementation of the OOFS library and the various daemons which make it useful is relatively straightforward. Calls which take pathnames pass them to a central routine which examines a pathname component by component checking for daemon attachments⁷. If a daemon is found, it is invoked (if it is not already running). Communication with the daemon is by default with a pair of conventional pipes, one for reading and one for writing, but it is also possible to connect to an already-running daemon at a named UNIX-domain socket. (Allowing daemons to persist across client invocations eliminates multiple time-consuming remote server connection interactions, and can also simplify caching.)

Most calls (`stat`, `rename`, `unlink`, etc.) pass a single request to the daemon and return its response to the caller. The `open` call, however, allocates an OOFS open file structure containing a pointer to the daemon, and returns an integer file descriptor which, when passed in again by the caller in a `read`, `write`, or other I/O call, will be recognized as an OOFS-handled file descriptor and will instigate a daemon transaction.

Calls involving pathnames and file descriptors not associated with OOFS daemons are of course passed on to the corresponding UNIX kernel system calls for conventional interpretation.

In order to support a shell linked against the OOFS libraries, and to permit shell redirection to and from daemon-handled files, special handling is necessary during `fork` and `exec` calls, which the OOFS library also intercepts. Before an `exec`, current directory and open file state information is saved in the environment variable `OOFSCONTEXT` so that the copy of the OOFS library in the invoked program can recover it. Negotiations are performed before and after a `fork` so that the parent and child (which might otherwise share communication paths to a single daemon) will not interfere with each other, in particular to insure that one will not be able to close a file which should remain open in the other. (Tichy describes an alternate solution to this problem in [Tichy 84]; the problem is intriguingly similar to one uncovered when an early version of UNIX first implemented multitasking [Ritchie 84a].)

The OOFS library (the part which is linked in with client applications) is currently written in approximately 4800 lines⁸ of C, which compiles to 27 KB of object code. The three daemons discussed in this paper (`ftp`, `gopher`, and `WWW`) are written in approximately 4000, 2000, and 3000 lines of C, respectively, and have executables of size 57, 40, and 48 KB. (These sizes are all for a Sun 4; object sizes for non-RISC processors are somewhat smaller, while executable sizes for systems without shared libraries are somewhat larger.)

6.1. Daemon Interface Protocol

The communication protocol between an application—specifically, the OOFS library linked in with an application—and an OOFS daemon is based on simple text lines, for simplicity and ease of debugging. Each request is a line of the form

```
op modifiers wrsize rdsiz [args]
```

where *op* is a string representing the operation requested, *modifiers* is a string requesting optional kinds of behavior (none are yet defined), *wrsize* is the number (represented as a string) of bytes of data which accompany the request, *rdsiz* is the number of bytes of returned data the caller is prepared to accept, and *args* is a list of zero or more operands specific to the particular operation. An escape mechanism permits operands which are pathnames to contain spaces or other special characters, if necessary.

7. And symbolic links, which must be specially handled.

8. As reported by wc; these are not SLOC.

Each operation results in a return line of the form

```
status retval rdcunt [string]
```

where *status* is a number (again represented as a string) indicating the success or failure of the operation (including conditions such as "operation not supported"), *retval* is the value to be returned to the caller (or, for unsuccessful calls, the value to be placed in `errno`), *rdcount* is the number of bytes of data which follow, and *string* is an (optional) string encoding miscellaneous, possibly operation-specific, information. Currently the string is used to encode the return value of *seek* operations, since *retval* is limited to `int`-sized values; it will eventually also be used to encode error information at a higher level of detail than can be expressed in `errno` values.

When a request must send some data (i.e. a write-like request), and when data must be returned (from a read-like request), the data immediately follows the request or response line; the counts which appear in the request and response lines inform the receiving end how many data bytes it should read from the pipe.

The basic list of operations, most of which are supported by most daemons, includes the following requests:

chdir	mkdir	read/dir	start
chmod	open	read/dir/stat	stat
chown	open/dir	rename	unlink
close	quit	rmdir	utime
fork	read	seek	write

The *start* request is the first operation sent to a newly-invoked daemon; it verifies successful daemon startup and performs protocol version negotiation. *quit* is similarly used to shut down a daemon. The other operations have functions suggested by the UNIX system calls after which they are named; a few have unusual behavior:

The *stat* operation accepts either a pathname or an open file descriptor; it thus supports both the *stat* and *fstat* system calls.

open/dir announces intent to read a file as a directory; *read/dir* reads an open file as a directory and returns filenames; *read/dir/stat* reads a directory and returns filename and selected stat information at the same time. (The data returned by the two *read/dir* variants is of course in a filesystem-independent format.)

fork notifies the daemon that the client has forked and that it must be more careful about honoring *close* requests.

It is not immediately fatal for a daemon not to support an operation; when a daemon cannot meaningfully perform some operation, the OOFS library simply returns `-1` to the calling program, with `errno` set to `EIO` or `EOPNOTSUPP`.

It will be noted that the basic protocol is synchronous; it is also fairly stateful. Extensions to the protocol are planned in order to support asynchronous operation; it is also intended that all descriptor-based operators (*read*, *write*, etc.) alternatively accept pathnames and offsets, to better support stateless operation.

6.2. System Call Interception

A user-mode library such as OOFS which intercepts system calls faces a mildly-tricky problem at link time: it wishes to provide entry points with names such as `_open`, `_read`, `_write`, etc., while also calling actual UNIX system calls with the same names. (This is, of course, a simple inheritance problem.) Numerous solutions to this problem can be imagined; the OOFS library as currently implemented uses one of two.

The library contains entry points named `oofsopen`, `oofsread`, etc. (i.e. the conventional system call names, prefixed with “`oofs`”). The actual linking strategy depends on the I/O calls being made by the application:

1. An application that uses the `stdio` package exclusively is linked against a reimplementation of the `stdio` library [Summit 89] which is based on the Oofs routines rather than the standard system calls.
2. An application that uses system calls directly is recompiled with invocation-line preprocessor defines of the form `-Dopen=oofsopen` (etc.) in effect.

It would also be quite possible (and preferable) to provide a variant version of a dynamically-linked `libc.a`, or to use a UNIX kernel which provides well-defined support for system call interception, in either case eliminating tedious recompilation and/or relinking. (Jones discusses several relevant aspects of system call interception in [Jones 93]. Other investigators have demonstrated the feasibility of “intercepting” filesystem-related calls by implementing specialized NFS daemons, or using the automounter interface.)

6.3. Daemon Implementation

Writing a simple, read-only daemon (i.e. one which can support, say, the `cat` program) is surprisingly easy; arranging for a daemon to map or simulate all UNIX filesystem semantics expected by any program is of course arbitrarily hard. Without going into too much detail, this section lists some of the difficulties (and surprises) encountered while implementing the daemons mentioned in this paper.

A daemon must decide whether it will read or write the remote item on-the-fly as the client issues *read* and *write* requests, or whether it will perform I/O to and from a local temporary file, copying the entire file at once when the file is opened (for reading) or closed (after writing). On-the-fly I/O can both reduce overhead and provide lower startup latency: the *open* and the first *reads* may return almost immediately. I/O to and from a local temporary file, on the other hand, allows random access and simultaneous access of multiple files. (If files are to be cached, the use of a local temporary file is implied in any case.) A hybrid scheme, which builds a temporary file incrementally while performing on-the-fly I/O, is also possible. However, it is not easy for a daemon to decide which of these transfer models to use, particularly because it does not have all the information it could use in making the decision. For example, UNIX I/O semantics do not provide an indication at *open* time of whether I/O will be sequential or random access.

Some protocols (notably `ftp`, and also of course many foreign filesystems) distinguish between text and binary files. Again, it is difficult for the daemon to decide which mode to use without information which UNIX programs are—quite happily—not accustomed to providing. The Oofs `ftp` daemon addresses this problem by interpreting special syntax in the pathname; Cate describes another solution in [Cate 92].

It is notoriously difficult to map error conditions from any new device or protocol onto the relatively small, fixed set of UNIX `errno` values.

It is in general difficult or impossible to fill in all of the fields in the `stat` structure when a daemon performs a *stat* request on a piece of data which is not really a file. (The `st_mtime` and `st_ino` fields are particularly troublesome.) When these fields are not or cannot be filled in appropriately, some applications may misbehave.⁹ Even when troublesome fields can be filled in, deriving values for them may be expensive, which is unfortunate if an expensively-derived field is not actually needed by the caller.

Eventual extensions to Oofs may provide workarounds for some of these difficulties, such as: extra, optional tuning parameters to be specified at *open* time;¹⁰ an extended *perror* mechanism; and an indication at the time of a *stat* call of which fields are needed by the caller and which are being reliably returned. Any use of these extensions by applications, however, would obviously be nonstandard, nontransparent, and not universal.

9. `find(1)` has perhaps the most pressing requirements for accurate `stat` values, but even such lowly tools as `mv`, `cp`, and `diff` typically inspect `st_dev` and `st_ino` to determine whether two files are identical.

10. Such parameters would not represent abandonment of UNIX’s typeless filesystem, but rather acknowledgement that foreign systems, with which interoperation is desired, are not always so enlightened.

7. Performance

It is difficult to provide precise measurements of the performance of this system in a meaningful way. There are certainly three areas in which the use of the OOFS scheme, and its associated daemons, potentially degrades performance. First, all pathname lookups are complicated by the necessity of checking for the auxiliary files which record daemon attachment. Second, invoking a daemon necessarily involves `fork` and `exec` overhead. Finally, in general, all data read and written by the client process passes through the pipe to the daemon, increasing overhead.

There are, however, compensating advantages of the scheme. The separate daemon process, though it necessitates extra IPC, is nevertheless a second process: if the I/O is at all complicated, the client may benefit by having the I/O offloaded to a second process. In any case, when a remote network service is involved, any extra IPC overhead on the local machine is likely to be swamped by the unavoidable network I/O.

To assess at least a few performance aspects quantitatively, three tests were performed, both with and without using OOFS.

The first test measures the time to `stat` a file; differences reflect the extra pathname processing, auxiliary file lookup, and daemon invocation. This test was performed in three ways: without using the OOFS library at all; with OOFS but on a file without an attached daemon; and with OOFS on a file with a "pass-through" daemon, which passes requests back to the local filesystem. The second case pays the pathname processing and auxiliary file lookup penalty; the third case additionally suffers daemon invocation overhead.

The second test measures the time to `cat` a 1 MB file, both with a conventional `cat` and an OOFS-aware `cat` plus the pass-through daemon mentioned above. Differences reflect both name lookup and IPC overhead.

The third test compares the time required to `ftp` a 1.5 MB file as opposed to copying it with an OOFS-aware `cp` and the OOFS `ftp` daemon. (The OOFS-aware `cp` is at an additional disadvantage since it performs several extra `stat` operations, necessitating additional remote `ftp` server interactions.)

The tests were performed on a lightly loaded Sun 4/280 running Sun/OS 4.1.3. For the tests involving OOFS, approximate user and system times for the daemon process are also presented. The data appear in Table 1.

	real	user	system	user, daemon	system, daemon
Test 1 (<code>stat</code>)					
no OOFS	0.0003	0.00002	0.0003	—	—
OOFS, no daemon	0.0037	0.0008	0.0016	—	—
OOFS with daemon	0.25	0.041	0.12	0.0019	0.0009
Test 2 (<code>cat</code> 1 MB file)					
no OOFS	13.7	5.1	0.7	—	—
OOFS	23.5	6.8	4.5	1.4	2.0
Test 3 (<code>ftp</code> 1.5 MB file)					
<code>ftp</code> (no OOFS)	90.1	0.2	2.5	—	—
<code>cp</code> plus OOFS	110.3	4.5	11.3	2.7	8.0

Table 1
Performance Comparisons (all times in seconds)

There is an obvious performance degradation when the OOFS library is in use, although it should be noted that this is a prototype implementation against which no serious optimization attempts have yet been made.

No attempt has been made to measure performance of the more interactive operations involving the gopher and WWW daemons.

8. Open Issues and Future Work

The core OOFS mechanisms could use improvements in several areas. The library needs to be merged with an established, kernel-resident system call or I/O interception scheme, or at least made to work as a dynamically-linkable run-time library. A well-defined inheritance mechanism would cement its reputation as an object-oriented filesystem and provide support for such situations as remotely-mapped self-uncompressing files. Ideally, daemon attachment would be recorded in the inode; to do so would obviously require both kernel modifications and extensions to the on-disk inode structure.

This project is one of those many that presume openness and trust; concerns of security and authentication have been secondary. Although the daemon interface protocol has a few authentication hooks, they are not really implemented in practice by the existing daemons. (Obviously, when a daemon is providing access to a public resource, authentication is a non-issue; the daemon doesn't really care who its client is.)

The filesystem daemon scheme provides a fertile potential bed for the introduction of Trojan Horses and other mayhem. (Having a program fire up simply because a file is accessed is a cracker's dream come true.) On a timesharing system, if many users are using OOFS-aware applications, it may be appropriate to ensure that daemons run as their authors, and not as their invokers.

When daemons are attached in relatively few places (the applications discussed in this paper involve only top-level pseudodirectories such as /ftp and /gopher), it is practical to manually maintain the auxiliary files which record daemon attachment. If heavier use of the scheme were to be made, it would be important to implement locking or other automated handling of the auxiliary files, to prevent conflicts, and to support automated updating of daemon attachments when files are renamed or deleted.

The attempt to map non-filesystem-oriented information services such as gopher and WWW onto a simulated filesystem has revealed a few things that the designers of such protocols could do to make the mapping task easier and more meaningful. Specifically, it would be beneficial (for any automated use, not just for OOFS daemons) if a protocol could provide:

1. Useful distinctions between error conditions (e.g. "mal-formed request" vs. "item not found" vs. "permission denied" vs. "unexpected I/O error");
2. A means of checking for the presence of an item, or returning status information about it, without retrieving its contents;
3. Short names for items (in addition to any implicit indices or links);
4. Dates and/or modification times for items; and
5. A well-defined way to retrieve partial items, for example to read the second 1 KB block of a 100 KB item.

The preceding wish list is in increasing order of difficulty, and decreasing order of likelihood. OOFS is able to proceed without any of these features, and many protocols may be utterly unable to provide them, but where possible their availability would make filesystem emulation considerably more seamless.

9. Comparison to Other Work

Remote/networked/distributed filesystems have been implemented and described many times; examples are the Newcastle Connection [Brownbridge 82], IBIS [Tichy 84], RFS [Rifkin 86], the Andrew File System [Howard 88], and NFS [Sun 89]. OOFS is more general, intended to allow arbitrary processing during file access; accessing remote filesystems over a network is but one obvious application.

Several systems have supported heterogeneous filesystems (in part in support of networked file systems, as above) by inserting a level of indirection at the filesystem interface: examples are Sun's Vnodes [Kleiman 86] and the Version 8 typed file system [Weinberger 84, Rago 90].

Inserting extra functionality at the system call interface is a process which appears in several guises; many of the systems described in this section implement their extensions in this way. Jones discusses several aspects of system call extension and provides an excellent bibliography in [Jones 93]; another implementation is described in [Krell 92].

The idea of attaching arbitrary processing modules to certain I/O streams is not new; it is central to Research UNIX's streams [Ritchie 84b] and to apollo's DOMAIN system [Rees 86]. What this paper calls "filesystem daemons" have been described and implemented several times: one implementation is Bershad and Pinkerton's "Watchdogs" [Bershad 88]; a related idea is implemented by Eggert and Parker's IFS in [Eggert 93].

OOFS, then, is not terribly unique: it shares with Watchdogs and IFS the ability to do arbitrary processing, with per-file (as opposed to per-filesystem) granularity. It shares with the Newcastle Connection, IBIS, and IFS a user-mode implementation which requires no kernel modifications. One significant feature of the OOFS scheme is that it is designed to work with daemons which provide both more and less than the canonical level of processing: some daemons are barely able to provide a minimal simulation of filesystem semantics, but some daemons are able to support extended operations unheard of in conventional filesystems.

Current networking issues of broad interest include the problems of resource discovery, resource naming, resource organization, and resource access. Resource discovery is the focus of archie [Emtage 92], gopher, and WWW. One attempt at a systematic approach to uniform resource naming in the face of heterogeneous protocols is the Uniform Resource Locator (URL) scheme [Berners-Lee 93]. Much ongoing research attacks the resource organization problem; the Prospero system [Neuman 89] provides an excellent example. The applications described in this paper are primarily directed at the resource access problem, although the abilities to browse heterogeneous resources using standard tools and target them with symbolic links provide some support for the discovery and organization problems. (Appropriate OOFS daemons, including those described in this paper, could also provide a tidy implementation of the URL scheme, in the form of a `/url` pseudodirectory containing subdirectories `ftp`, `http`, etc.)

10. Conclusions

This paper has described a mechanism whereby heterogeneous network information services can be integrated more-or-less transparently into a local filesystem, such that they can be accessed and manipulated using standard, familiar tools. The emulation is not perfect (not all information sources can be made to mimic all filesystem semantics), but the limitations of the emulation are definitely balanced by the advantages of being able to use standard tools. When tools and protocols share standard interfaces, they can be combined in arbitrarily powerful ways to solve problems not originally envisioned.

The integration depends, in this case, on making the filesystem interface (as embodied in the operating system calls `open`, `read`, etc.) a *rendezvous point* between network services and local utilities, such that services and utilities written at different times, under different sets of assumptions, with different immediate goals in mind, by different people, and without advance knowledge of each other, can nevertheless interoperate.

Acknowledgements

Thanks to Mark Brader, Stan Brown, Paul Eggert, Michael Jones, Jeff Mogul, and Melanie Summit for their comments on earlier drafts of this paper. Thanks to Robert Dinse at Eskimo North for providing the system on which most of this work was performed.

References

- [Anklesaria 93] F. Anklesaria *et al*, "F.Y.I. on the Internet Gopher Protocol," March, 1993, URL=ftp://boombox.micro.umn.edu/pub/gopher/gopher_protocol/DRAFT_Gopher_FYI_RFC.txt (also RFC-1436).
- [Berners-Lee 93] Tim Berners-Lee, "Uniform Resource Locators," Internet Draft, March, 1993, URL=<ftp://info.cern.ch/pub/ietf/url4.txt>.
- [Bershad 88] Brian N. Bershad and C. Brian Pinkerton, "Watchdogs—Extending the UNIX File System," *Computing Systems*, 1:2, 1988, pp. 169-188.
- [Brownbridge 82] D.R. Brownbridge, L.F. Marshall, and B. Randell, "The Newcastle Connection or UNIXes of the World Unite!," *Software—Practice and Experience*, 12, 1992, pp. 1147-1162.
- [Cate 92] Vincent Cate, "Alex—a Global Filesystem," in *Proceedings of the USENIX File Systems Workshop*, Ann Arbor, MI, 1992, pp. 1-11.
- [Davis 90] Franklin Davis *et al*, "WAIS Interface Protocol Prototype Functional Specification," April 23, 1990, URL=<ftp://quake.think.com/pub/wais/doc/protspec.txt>.
- [Eggert 93] Paul R. Eggert and D. Stott Parker, "File Systems in User Space," in *Proceedings of the Winter 1993 USENIX Conference*, San Diego.
- [Emtage 92] Alan Emtage and Peter Deutsch, "archie—An Electronic Directory Service for the Internet," in *Proceedings of the Winter 1992 USENIX Conference*, San Francisco, pp. 93-110.
- [Howard 88] J.H. Howard, "An Overview of the Andrew File System," in *Winter 1988 USENIX Conference Proceedings*, Dallas, February, 1988.
- [Jones 93] Michael B. Jones, "Interposition Agents: Transparently Interposing User Code at the System Interface," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Asheville, NC, December, 1993.
- [Kahle 89] Brewster Kahle, "Wide Area Information Server Concepts," November 3, 1989, URL=<ftp://quake.think.com/pub/wais/doc/wais-concepts.txt>.
- [Kleiman 86] S.R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," in USENIX Association, *Summer Conference Proceedings*, Atlanta, 1986, pp. 238-247.
- [Korn 90] David G. Korn and Eduardo Krell, "A New Dimension for the Unix File System," *Software—Practice and Experience*, 20:S1, June, 1990, pp. 19-34.
- [Krell 92] Eduardo Krell and Balachander Krishnamurthy, "COLA: Customized Overlaying," in *Proceedings of the Winter 1992 USENIX Conference*, San Francisco, pp. 3-7.
- [Krol 92] Ed Krol, *The Whole Internet User's Guide & Catalog*, O'Reilly & Associates, 1992, ISBN 1-56592-025-2.
- [Neuman 89] B. Clifford Neuman, "The Virtual System Model for Large Distributed Operating Systems," Technical Report 89-01-07, April, 1989, Department of Computer Science, University of Washington, Seattle.
- [Rago 90] Stephen Rago, "A Look at the Ninth Edition Network File System," in *UNIX Research System Papers*, Volume II, Saunders College Publishing, 1990, ISBN 0-03-047529-5.

- [Rees 86] Jim Rees, Paul H. Levine, Nathaniel Mishkin, and Paul J. Leach, "An Extensible I/O System," in USENIX Association, *Summer Conference Proceedings*, Atlanta, 1986, pp. 114-125.
- [Rifkin 86] Andrew P. Rifkin *et al*, "RFS Architectural Overview," in *USENIX Conference Proceedings*, Atlanta, GA, June, 1986.
- [Ritchie 84a] Dennis M. Ritchie, "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal*, 63:8, October, 1984, pp. 1897-1910.
- [Ritchie 84b] Dennis M. Ritchie, "The Evolution of the UNIX Time-sharing System," *AT&T Bell Laboratories Technical Journal*, 63:8, October, 1984, pp. 1577-1593.
- [Roome 92] W.D. Roome, "3DFS: A Time-Oriented File Server," in *Proceedings of the Winter 1992 USENIX Conference*, San Francisco, pp. 405-418.
- [Summit 89] Steve Summit, "A Reimplementation of the Standard I/O Package," unpublished.
- [Sun 89] Sun Microsystems, Inc., "NFS: Network File System Protocol Specification," Internet RFC-1094, March, 1989.
- [Tichy 84] Walter F. Tichy and Zuwang Ruan, "Towards a Distributed File System," in USENIX Association/Software Tools Users Group, *Summer Conference Proceedings*, Salt Lake City, 1984, pp. 87-97.
- [Warnock 84] Robert P. Warnock III, "User-Mode Development of Hardware and Kernel Software," in USENIX Association/Software Tools Users Group, *Summer Conference Proceedings*, Salt Lake City, 1984, pp. 224-226.
- [Weinberger 84] P.J. Weinberger, "The Version 8 Network File System" (abstract), in USENIX Association/Software Tools Users Group, *Summer Conference Proceedings*, Salt Lake City, 1984, p. 86.
- [WWW 93] "Protocol for the Retrieval and Manipulation of Textual and Hypermedia Information," June, 1993, URL=<ftp://info.cern.ch/pub/www/doc/http-spec.ps> .

Author Information

Steve Summit attended the Massachusetts Institute of Technology from 1979 to 1983, receiving a Bachelor of Science degree in Electrical Engineering and Computer Science. He has worked as a software engineer since then, presently as an independent consultant. His interests and specialties are too undifferentiated, and his mistrust of buzzwords too complete, to attempt to list them here; this paper is however not unrepresentative. He can be reached at scs@eskimo.com .

Concert/C: A Language for Distributed Programming

*Joshua S. Auerbach
Ajei S. Gopal*

*Arthur P. Goldberg
Mark T. Kennedy
James R. Russell*

*Germán S. Goldszmidt
Josyula R. Rao*

*IBM Thomas J. Watson Research Center
P. O. Box 704, Yorktown Heights, NY 10598*

Abstract

Concert/C is a new language for distributed C programming that extends ANSI C to support distribution and process dynamics. Concert/C provides the ability to create and terminate processes, connect them together, and communicate among them. It supports transparent remote function calls (RPC) and asynchronous messages. Interprocess communications interfaces are typed in Concert/C, and type correctness is checked at compile time wherever possible, otherwise at runtime. All C data types, including complex data structures containing pointers and aliases, can be transmitted in RPCs.

Concert/C programs run on a heterogeneous set of machine architectures and operating systems and communicate over multiple RPC and messaging protocols. The current Concert/C implementation runs on AIX 3.2¹, SunOS 4.1, Solaris 2.2 and OS/2 2.1, and communicates over Sun RPC, OSF/DCE and UDP multicast. Several groups inside and outside IBM are actively using Concert/C, and it is available via anonymous ftp from `software.watson.ibm.com:/pub/concert`.

1 Introduction

This paper describes Concert/C, a new language for distributed programming. We describe the basic Concert/C features needed to write significant Concert/C programs. We also show example Concert/C programs, and present comparisons with traditional RPC technology to motivate our design choices.

The objective of the Concert project is to develop tools to improve the productivity of people writing distributed programs for use in today's rich networking environments. Since most workstation and PC programmers write in C, we chose to extend C. We added a minimal set of new concepts, types, and operators, and reused existing C features whenever possible, so that a C programmer could learn Concert/C easily. In addition, we made Concert/C programs link-compatible with existing C object code to avoid sweeping recompilation of legacy code.

In big organizations, most LANs connect a heterogeneous set of machines, including personal computers running DOS, Windows, OS/2 and NetWare, Macintoshes, and workstations running different flavors of Unix. A variety of protocols run over these nets, including SNA, TCP/IP, NetBios and SPX/IPX. We designed Concert/C to exist in this environment. Our compiler is portable to any system with an ANSI C compiler, and our "multi-protocol" run-time can communicate over a heterogeneous set of protocols.

We chose to implement Concert/C as a language, rather than a set of library functions like PVM [33] or a set of library functions plus a stub compiler like SUN ONC assisted by tools such as `rpcgen` [32]. We made this decision because language-integrated support for distributed programming can offer a higher level of functionality than other approaches, as we demonstrate in section 3.

¹ All trademarks appearing in this paper are recognized registered trademarks of their respective companies.

2 The Concert/C Language

Concert/C [3, 4] is a superset of ANSI C. Like an ANSI C program, a Concert/C program is built from separately compiled source files combined in a link step. Constructing it differs from constructing an ANSI C program only in the use of the Concert/C compiler (`ccc`) instead of an ANSI C compiler (e.g. `cc`) for those source files which contain Concert extensions.

Concert/C is fully link-compatible with ANSI C, so only a small fraction of the application will typically need to be compiled with `ccc`. `ccc` is actually implemented as a preprocessor using ANSI C as its back end, which enhances portability, helps ensure object code compatibility, and allows ANSI C debuggers to be used. A Concert/C program is a normal executable, and may be placed into execution by all the usual means.

The design of Concert/C's extensions exploits the fact that `ccc` reads application source files containing both interface declarations *and* program logic. Although `ccc` does not transform the program logic (for example, Concert/C has no control-flow primitives not taken directly from ANSI C), it can find many errors and automate certain functions which it could not if it saw only interface declarations. We call this a *single-translator* approach because, `ccc` can translate anything that `cc` can (in practice, it does not always need to).

Our single-translator approach contrasts with that of "add-on" RPC packages (such as SUN ONC, Apollo NCS, or OSF DCE), which may be characterized as a *two-translators* approach. In that approach the C compiler reads all of the program's original executable logic while an IDL translator or stub compiler (for example, `rpcgen`), reads interface definitions. Neither translator can read the other's source files, so the original executable logic is joined with the stub compiler's "canned" logic for marshalling and demarshalling of messages only at link time. This limits the extent to which the tool can help the programmer.

We describe the features and language extensions of Concert/C below. These fall into six general categories. While add-on packages could theoretically provide some help in each of the areas listed below, most provide only RPC, a single method for exchanging bindings, and (in the more recent ones) a strategy for demultiplexing requests based on locally concurrent threads. Some provide messaging, but none that we know of provides any help with process management. A complete facility for distributed computing must cover all these areas.

Despite the fact that Concert/C uses a single-translator approach, not every feature is provided as a language extension; language extensions are introduced only where there is significant "mileage" to be gained. We publish explicit runtime library calls, or use preprocessor macros, for other cases where a language extension is not required. By convention, Concert/C language keywords have simple mnemonic names while library functions and macros have names beginning with `cn_`. The examples in section 3 will illustrate these in more detail.

1. *Transparent Remote Function Calls:* Concert/C transparently "overloads" the C function call construct to embrace remote procedure call (RPC). If there are several remote instances of the same interface, Concert/C uses different function pointer *values* as a way of determining to which instance a call is to be directed, rather than using a "handle" parameter for this purpose. In general, a pointer to a remote function serves as a *binding*, conveying the capability to invoke that function.

Placing a function in the **port** or **initial port** storage class causes a message queue to be associated with the function, enabling it to be called remotely.

2. *Asynchronous Messaging:* Concert/C adds one-way, asynchronous, queued messaging to the language. By analogy to function pointers, a pointer to a remote queue serves as a binding, conveying the capability to enqueue a message.

The `receiveport { message-type }` type constructor defines a queue of messages of type *message-type* (which may be any Concert/C type). The operation `send(binding, message)` sends *message* to the queue to which *binding* refers, and the blocking operation `receive(port, message-ref)` moves a message from a queue into the variable pointed to by *message-ref*.

3. *Process Management*: Concert/C adds process management to the language. The new types `process` and `program` describe Concert/C processes and programs. A Concert/C program can start another Concert/C program executing on the same or a different machine using the `create(program, binding-ref)` operation. The *binding-ref* argument is a pointer to a binding variable in the parent, which is initialized by `create` to provide a logical connection to the child (specifically, a pointer to the initial port in the child) which is immediately usable for either RPC or messaging. `Create` also returns a value of type `process`, which can later be used by the `terminate` operation to kill the created process.
4. *Distributed Linking*: Concert/C provides an extensible set of "External Binding Facilities" (EBFs) which can store and retrieve Concert/C remote pointers to and from external media of various kinds. Many popular methods of finding components in a distributed system are supported in this way, including "well-known" ports (like the ones listed in `/etc/services`), the SUN ONC portmapper, shared files, strings, and the OSF DCE cell-directory service. This becomes a vehicle both for evolving the connectivity of a Concert/C application, and also for interoperating with popular RPC-based components not written in Concert/C. When used in the latter way, EBFs transparently map between Concert/C's use of "remote pointers" to express logical connectivity and whatever method is used by the other component. EBFs are provided both in executable macro form and in a convenient declarative form which is executed automatically at program start time. In some cases, use of declarative EBFs removes the need to write a main function.
5. *Demultiplexing*: Concert/C provides the necessary demultiplexing primitives to permit a server to make progress in the face of arbitrary request arrival orders. Included in this category is the ability either to deal with an RPC in a "rendezvous" style as a procedure call or decompose it into "request" and "response". The blocking operation `accept(function-port-list)` automatically handles a call to one of the function ports in the list when it arrives. The blocking operation `select(queue-list)` and the non-blocking operation `poll(queue-list)` return the index of a queue in *queue-list* that has a message (`poll` returns 0 if no queue does). These can be used to schedule the message receive. The new type `cmsg{function-type}` holds the "request" message associated with a remote call. The operation `reply` issues the "response" to such a message, thus completing a decomposed call.
6. *Communication of Complex Data Types*: Concert/C permits arbitrary C data types, including scalars, arrays, unions, and data structures built with pointers and aliases, to be transmitted in interprocess communications. If the data being transmitted (as an argument to an RPC, or as a message) contains pointers, then marshalling will make a *deep copy* – following the pointers², and the pointers in the data they point to, and so on, copying the entire data structure for transmission. Marshalling and demarshalling some data types requires more information about the data than standard C type declarations provide. The Concert/C programmer can provide additional information by *annotating* the IPC data. Annotations can be placed on declarations of data types communicated between processes, and declarations and definitions of ports and bindings. The annotations are also called *attributes*.

The use of attributes is unnecessary for "simple enough" functions. A function is "simple enough" if it uses no unions, arrays are limited to character strings, pointers don't introduce any actual aliases, and parameters pass information in the "obvious" directions ("in only" for non-pointers, both directions for pointer-indirected arguments).

Attributes add several kinds of information to communications.

- *Data description* attributes expand on C's description of data types. For example, they indicate the length of arrays, the presence of null-terminated character strings, and the case of unions.
- *Communication direction* attributes indicate the transmission direction of remote function call parameters. An *in* argument is transmitted from the caller to the callee, an *out* argument from the callee to the caller, and an *in,out* argument is transmitted both ways.

²However, the annotation `opaque` can instruct the marshaller to not follow a pointer.

Concert/C Attributes

<i>Attribute</i>	<i>Meaning</i>
How much memory is allocated to hold an array?	
max_length(i)	memory for an array of i elements is allocated.
Which direction is an argument transmitted?	
in	argument is transmitted from caller to callee.
out	argument is transmitted from callee to caller.
in,out	argument is transmitted from caller to callee and from callee to caller.
What case is a union?	
switch_is(i)	identifier i indicates the case of the union.
switch_type(type_specifier)	type_specifier indicates the type of the union discriminant.
case(case_list)	case_list identifies the case the union is in when the discriminant has a value appearing in case_list.
Can a pointer reference an object that other pointers reference?	
aliased	yes. marshaller detects aliases. aliased data at sending end becomes aliased data at the receiving end.
unique	no. marshaller does not detect aliases.
How big an array should be transmitted?	
length(i)	an array has i elements.
string	following the C character string convention, the array is terminated by a null character.
Which allocated memory is freed by the marshaller?	
keep(caller)	memory in caller or sender is not freed.
keep(callee)	memory allocated in callee is not freed.
discard(caller)	memory in caller or sender is freed.
discard(callee)	memory allocated in callee is freed following return.
Might a pointer be NULL?	
optional	yes. check for NULL pointer when marshalling.
required	no. marshaller assumes pointer references data to transmit.

Figure 1: Concert/C attributes.

- *Storage retention* attributes help control the marshaller's use of heap storage. Demarshalling allocates heap memory to store some IPC data. Storage retention attributes control whether the marshaller de-allocates these data.

Attributes appear in the syntax as a type-qualifier, like **const** and **volatile**. They are given in a comma separated list contained in square brackets: **[attribute1, attribute2,...]**

The Concert/C attributes are summarized in figure 1.

3 Examples and Comparisons

In this section, we present some detailed examples of the Concert/C language. We will show how each extension is accomplished, and contrast it with what would be the case if we were limited to the two-translators approach followed by add-on RPC packages. To make the discussion concrete, we will use a single programming example (which we will elaborate as we go along), showing how Concert/C does things, and also how the same thing would need to be done using public domain SUN ONC with `rpcgen`.

We chose this technology for the comparison because it is familiar and widely available, so most readers will be able to try similar examples for themselves. In some areas, other packages (such as NCS or DCE) may be more functional than public domain SUN ONC; we will try to point these out. However, the limitations on which we will focus most heavily are intrinsic to the two-translators approach, and are true of *all* add-on RPC packages.

3.1 Transparent RPC, Single Server, Fixed Relationship

Our simplest example starts out with one client program and one server program. In the usual RPC style, we will express the server first as a subroutine to be called (perhaps) remotely. Its purpose is to provide either an error code (if it fails) or (if it succeeds) a linked list of structures which constitutes a report on machine resource utilization by user.

We started with this example because the two-translators approach does reasonably well with it. In fact, the two-translators approach tends to do well as long as a client must interact with only one server of a given type. Even here, however, Concert/C has some significant advantages, stemming from the programmer's ability to reason about types using one language rather than two, and from the simplified mechanics of program construction.

Conceiving the example first as a nondistributed, ANSI C program, its interface definition, contained in a header file (`example1.h`), might look something like the following.

```
enum status { success, failure };
enum reason { not_authorized, system_error };
struct user { /* list of users and their utilization, expressed as a list */
    struct user * next;
    char * name;
    unsigned long cpu;
    unsigned long memory;
    unsigned long disk;
};
union result {
    struct user * userlist;
    enum reason why_failure;
};
enum status get_utilization(union result *answer);
```

A client should look something like this.

```
#include "example1.h"
main()
{
    union result a;
    switch(get_utilization(&a))
    {
        case success:
            /* display result list anchored at a.user_list */
            break;
        case failure:
            /* report failure reason from a.why_failure */
            }
    }
```

The server subroutine would look something like the following (omitting all the purely local logic).

```
#include "example1.h"
```

```

enum status get_utilization(union result * answer)
{
    /* try to gather the statistics (never mind how) */
    if (it_worked) {
        answer->userlist = what_we_gathered;
        return success;
    }
    else {
        answer->why_failure = why_it_failed;
        return failure;
    }
}

```

In Concert/C, to make this a distributed client-server application, we just take the header file, and annotate the definition of union result and the type of function get_utilization.

```

[switch_type(enum status)]
union result {
    [case(success)]
    struct user * userlist;
    [case(failure)]
    enum reason why_failure;
};
typedef enum status get_utilization_t(
    [out, switch_is(return)] union result *answer
);

```

We also changed the function definition to a type definition; this was not strictly necessary but is a convenience for later.

It is not necessary to recompile either the original client or the original server subroutine with ccc. Instead, a server main procedure is obtained by using ccc to compile the following separate Concert/C source file.

```

#include "example1.h"
extern port get_utilization_t get_utilization;
[[
    use shared_file;
    export get_utilization {to "somefilename"};
    automain;
]]

```

The first line is simply a declaration of the function get_utilization in terms of its type. The port keyword is a Concert/C extension which informs the compiler that the function in question might be called remotely. The double brackets set off a section containing the declarative form of EBF syntax. The use statement indicates which EBF is to be employed (the "shared file" EBF in this case), export invokes some EBF-specific logic to export a function or message queue pointer to an external medium, and automain requests an automatic main function which loops forever listening for requests to all exported functions.

In the client, the simplest way to bind client and server together is to provide the matching declaration and EBF syntax, again in a separate ccc source file.

```

#include "example1.h"
get_utilization_t get_utilization;
[[

```



```

    use shared_file;
    import get_utilization {from "somefilename"};
}

```

Note that we didn't use the `port` keyword here, because this module is not going to have it's `get_utilization` called remotely from any other module; rather, `get_utilization` is to be resolved by importing from a shared file. Putting the Concert/C client extensions in a separate source file to avoid recompilation was only a suggestion. The Concert/C client and server extensions could just as easily have been placed in the same source files with the original client and server source.

In comparison, to provide the same functionality using the two-translators approach, one needs to supply a separate IDL file. The following example shows one for `rpcgen`. While it resembles the original header file, note that it differs from ANSI C syntax, and thus the programmer must construct it separately.

```

enum status { success, failure };
enum reason { not_authorized, system_error };
struct user {
    user * next;
    string name<>;
    unsigned long cpu;
    unsigned long memory;
    unsigned long disk;
};
union result switch (status discrim) {
    case success:
        user * userlist;
    case failure:
        reason why_failure;
};

program GET_U_PROG {
    version GET_U_VERS {
        result get_utilization (void) = 1;
    } = 1;
} = 0x31234567;

```

In contrast to Concert/C, the correspondence of IDL types in the IDL file to C types in the C source is no longer exact. The enums retained their ANSI C syntax, but we had to deviate from ANSI C when we got to the struct and the union. We needed the special type string. We also had to assign a *name* (discrim) as well as a type (status) to the union discriminant, because `rpcgen` will actually generate a structure with a union and the union's discriminant encapsulated inside it. The type name `result` will actually refer to that structure, not the union. In order to use what `rpcgen` generates, we have to understand that it has done this transformation (in general, the generated interface header file has to be read). We were forced to relinquish the use of the identifiers specified for the enum and struct tags since `rpcgen` has automatically declared them as typedef names. Finally, we have had to accept a compromise both on the name (it becomes `get_utilization_1`, incorporating the version number) and on the exact type signature of the remote function (it needs to return a single output and take into account the "encapsulated" form of union which `rpcgen` generates). As rewritten, it looks something like the following.

```

#include "example1.h" /* generated by rpcgen */
result * get_utilization_1(void)
{
    static result answer;

```



```

/* try to gather the statistics (never mind how) */
if (it_worked) {
    answer.result_u.userlist = what_we_gathered;
    answer.discrim = success;
}
else {
    answer.result_u.why_failure = why_it_failed;
    answer.discrim = failure;
}
return & answer;
}

```

The change was a modest one, but that is not our point. The programmer had to understand a number of things about the correspondence between the language of the stub compiler and the programming language in order to get the program into the correct form. This need for mental translation is avoided when the interface declaration language and programming language are more tightly integrated to begin with, as they are with a single-translator approach.

These difficulties are inherent in the two-translator approach. Other stub compilers may do better than `rpcgen` in making some of the steps convenient, or in providing more overall expressiveness in the IDL (for example, OSF DCE's IDL is sufficiently expressive to cover most things that C programmers want to do), but *all* add-on RPC packages require programmers to do some mapping between two *different* type systems. Also, all such packages require treating the IDL file as something requiring a special compilation step; it is not directly usable as include file. Mechanically, the stub compiler produces many outputs (a header file, and several source files, some of which need to be incorporated into client and some into server). The Concert/C case had none of those complexities, although it did require some declaration to be done (of course).

It happens that SUN ONC with `rpcgen` also requires an explicit "binding" call (and subsequent error handling) in the client program in order to produce a "client RPC handle." It also requires that that handle be passed to the client stub procedure, making the RPC considerably less transparent. We won't show that here, since other add-on RPC packages, such as OSF DCE, succeed in hiding this particular source of complexity (about as well as Concert/C does) for this simple one-server case. However, in the next section we will see that a small change to the problem will force explicit binding and some form of "RPC handle" to be used in all known RPC tools that use the two-translators approach.

A server main procedure is generated automatically by `rpcgen` just as it is for simple-enough cases with Concert/C.

Before leaving this simple example, we consider a variation: suppose that the server had already been written using SUN ONC and `rpcgen`, and we wish to call it from a Concert/C client. Concert/C does not require that the server be rewritten in Concert/C to accomplish this. To write the client, we recognize that the server was written using a technology that requires certain restricted forms for function signatures, and simply change our view to conform to its, by slightly modifying the Concert/C declaration.

```

struct result {
    enum status discrim;
    [switch_is(discrim)]
    union result {
        [case(success)]
        struct user * userlist;
        [case(failure)]
        enum reason why_failure;
    } u;
};
struct result * get_utilization(void);

```

Then, in the Concert/C client program we have the following EBF logic.³

```
cn_sunrpc_import(HOST, GET_U_PROG, GET_U_VERS,  
    GET_U_PROC, 0, "udp", get_utilization);
```

HOST needs to be given some value (the host on which the appropriate SUN ONC server resides). GET_U_PROG, GET_U_VERS, and GET_U_PROC are constants for the program number, version number, and procedure number assigned by rpcgen. The 0 indicates that the portmapper is to be used (a nonzero here would be taken as a well-known port) and the next argument indicates which of SUN ONC's two primary protocols are to be used. The last argument is, of course, a reference to the function which is being imported. After executing this EBF macro, we can call the imported function `get_utilization` just as before. A similar approach could be used for the dual of this variation: where the client is written using rpcgen and the server using Concert/C.

An EBF macro such as is illustrated here, although provided as a preprocessor macro backed by a library function, could actually not be supported in such a convenient form without our single-translator approach; that is, it requires language extensions underneath. In the expansion of this macro, the primitive language extensions `crt_marshall_plan(get_utilization)` and `crt_microstub(get_utilization)` are employed. The Concert/C compiler, which knows the type of function `get_utilization`, responds to the first of these directives by generating a table of anonymous stubs (called a "marshall plan"). This avoids requiring any static name to be assigned to such stubs in the event that the type of the last argument were to be a function pointer. The value of this will be seen in the next section.

Similarly, because the compiler "knows" that that `get_utilization` is a function and not a function pointer, it responds to the second directive by providing a function body (known as a "microstub") which adjusts for the difference in calling mechanics between the two. This permits the underlying library function to always produce a function pointer result, making it more general.

Two key points are illustrated by this variation. First, it is possible to have the advantages of Concert/C while still interoperating with the non-Concert world; we use EBFs to express this. Indeed, one can use the `sunrpc` and `osf_dce`⁴ EBFs in the same program, and interoperate with both SUN ONC and OSF DCE. Of course, to actually deliver interoperability with SUN ONC and OSF DCE components is also a challenge⁵ for protocol management in the runtime and in stub compilation; we deal with this aspect of Concert/C in [5]. Second, supporting dynamic EBFs is greatly aided by having a single-translator approach. The compiler was able to discover the need to do stub compilation, and do it without assigning static fixed names to stubs; no separate stub compilation was needed.

3.2 Transparent RPC, Multiple Servers, Dynamic Relationship

Now let's change the problem slightly, in order to see where the power of Concert/C really pays off. Suppose we want to write a client which will invoke multiple instances of the server subroutine described in the previous section, each of these to execute on a different machine. Suppose, further, that the list of machines is given on the command line, and so not known statically. And suppose, finally, that we have not pre-planned which machines will need to run these servers, and so we have not arranged for them to be started automatically by `inetd`, would not register them with the portmapper, etc. In short, we want to start them as needed ("on the fly") and terminate them when we are finished.

To do this in Concert/C, we would not have to change the header file at all (beyond what we did already in the previous section), and we would not need to change the server subroutine either. We would discard the server `ccc` source containing `automain`, and use, instead, the following variation.

```
#include "example1.h"
```

³In the present Concert/C implementation, we have only an executable macro form for the "sunrpc" EBF, which is why we don't use the declarative syntax; such a syntax may be added soon.

⁴Under development.

⁵There is no uniform mechanism for handling external events within a Unix process; nor is there an API for routing event streams to the various API's which wish to consume them like the X Windows library, or the Sun RPC or OSF DCE runtimes.

```

initial port get_utilization_t get_utilization;
main()
{
    accept(get_utilization);
}

```

The accept operator provides the "rendezvous" facility for Concert/C. The main program illustrated above accepts exactly one call to get_utilization and then exits. By making get_utilization the initial port, we tell the compiler that not only will it be called remotely, but the Concert/C process which creates this process (the "parent") will automatically get a pointer to it. We lost the automatic main, but the main which we had to write was trivial. With any of the various two-translators approaches, we would also have to write a main procedure for this case, and it would be far from trivial.

The new client, now a full-fledged Concert/C program, looks something like this.

```

/* somewhere: define the server program name as SERVER_NAME */
#include "example1.h"
#include "concert.h" /* get the cn_ library functions and macros */
main(int argc, char *argv[])
{
    char * host;
    get_utilization_t * get_utilization_p;
    union result a;
    program pgm = cn_prog_init();

    cn_prog_set_name(pgm, SERVER_NAME);
    while (host = *(++argv))
    {
        cn_prog_set_host(pgm, host);
        if (create(pgm, &get_utilization_p) == cn_noprocess)
        { /* deal with create error */ }
        switch(get_utilization_p(&a))
        { /* like original client */
        case success:
            /* display result list anchored at a.user_list */
            break;
        case failure:
            /* report failure reason from a.why_failure */
        }
    }
}

```

Recall that there are Concert/C extensions in the header file; these are still needed, of course. We have added two new ones. The create keyword provides a new primitive operator (create is not a library function, though its syntax mimics one). The program keyword provides a new data type, generally called a "program description." A program description is a complex reference to a program, designating the program's file name, the machine it is to run on, and various options for instantiating it. The various cn_prog_ library functions are used to initialize program descriptions.

The create primitive puts a program into execution, yielding a value of type process (called a "process handle") as its expression value, and possibly mutating its second argument, which will receive a pointer to the initial function or queue in the created process. This pointer can be used to communicate with the "child." If we had retained the process handle in this example, we could have later used it with the terminate operator to cancel the child's execution (by the "severest" available means). This was not necessary since the child program was written to terminate voluntarily after one call. The only use we made of the process handle in the example was to compare it to special process handle value cn_noprocess which would mean that no child was created (a richer exception handling facility exists in

Concert/C but is not discussed in this paper). We used the pointer to the child's initial function in order to make an RPC to the child.

What is illustrated here is the joint power of providing process dynamics along with communication, and also using function pointers (which are anonymous and re-assignable) to express remoteness. By using this approach, we were able to write a highly dynamic client which was not much more complicated than the original static client. Although it may not be obvious at first, a single-translator solution is needed to make a pointer-based approach work effectively. Because `create` is an operator and not a function, `ccc` is involved in analyzing the type of its second argument, and detects that it is a pointer to a pointer to a function of a particular type. The necessary "marshall plan" (anonymous stub table) is generated automatically and associated with each pointer value generated by successive calls to `create`.

In fact, only one physical marshall plan is generated; it is reused with different "ministubs." A Concert/C ministub captures the specific information needed to find a particular server instance containing a remote procedure or queue. It arranges to call the right stub from the right marshall plan with the right routing information. This frees the application from any need to manipulate "RPC handles" or other explicit binding information and keeps RPCs "transparent" even while directing them to specific servers.

With any two-translators approach, since information from the two translators is merged only at link time, a static name (visible to the linker) *must* be assigned to each generated stub. If a stub is to be generated for each function *type* to support an indefinite (and dynamically determined) number of instances of that type, the add-on RPC package has *no choice* but to use a "client RPC handle" or similar construct to discriminate among server instances, making RPC inherently less transparent and requiring the application to become involved in the binding process.

3.3 Parallelism, Messaging, Passing Bindings

A further elaboration of our sample program shows some other convenience features of Concert/C. Suppose that the gathering of statistics at the server was expected to take a long time, so that it was useful to run all the servers in parallel. We would, therefore, like to restructure our application so that, first, RPCs are made to all servers asking them to start gathering the statistics, and then, as they gather them, the servers send the statistics back, asynchronously, in parallel. A set of declarations which is reasonable for this purpose is the following.

```
enum status { success, not_authorized, system_error };
/* the above collapses what used to be 'status' and 'reason' */
struct report { /* report on one user at one machine */
    char * machine;
    char * name;
    unsigned long cpu;
    unsigned long memory;
    unsigned long disk;
};
/* the above is no longer a linked list, and a union is no longer used */
typedef receiveport {struct report} report_port;
typedef enum status start_gathering_t(report_port * answer);
```

Note that *no* Concert/C attribute lists are needed for this set of declarations. That is because no arrays (other than character strings), unions, aliased pointers, or parameters of uncertain directionality are employed. A new language extension is present, however, in the `receiveport` declaration. A `receiveport` is simply a queue of messages of any type. We also declare a function type designed to be the initial function of each server. This function is passed a `receiveport` pointer which will be kept by the server and used later to send results back. The initial function returns an `enum status` indicating whether the server is capable of carrying out the request.

The server would look something like this.

```
#include "example3.h"
```

```

report_port * answer_port;
initial port enum status start_gathering(report_port * answer)
{
    /* set up to gather statistics */
    if (/* can do it */)
    {
        answer_port = answer; /* save queue pointer */
        return success;
    }
    else
        /* return one of the failure codes */
}

main()
{
    struct report r;

    accept(start_gathering); /* one call to initial port */
    while (/* gathering statistics */)
    {
        /* gather record for a user in r */
        send(answer_port, r);
    }
    /* build special 'null user' record in r */
    send(answer_port, r); /* logical end of file */
}

```

The client to invoke this server would begin with the same logic as was used in the previous section, creating the servers and invoking their initial ports. However, as an argument to each call, it now passes the address of its receiveport called `answer_queue`. It would issue messages for those servers that couldn't perform the task, and count how many could. Then, it would do something like the following.

```

...
report_port answer_queue;
struct report r;
...
/* initial logic as described above */
...
while(server_count)
{
    receive(answer_queue, &r);
    if (/* real user element */)
        add_to_report(r);
    else /* logical end of file for a server */
        server_count--;
}
print_report();
}

```

Doing *something* similar in add-on RPC packages is sometimes possible, although many such packages do not provide asynchronous messaging. The most obvious way in which the single-translator helps here is that ccc can check whether the send and receive operations are type correct, finding errors sooner. In addition, the simple structure of the solution is greatly aided by being able to pass pointers to receiveports as first class values. These values are both *typed* and *anonymous*. The compiler generates

the necessary "marshall plans" (anonymous stubs) automatically, and makes sure they are associated with each value of type "pointer to receiveport" as it is being received. First class typed, anonymous pointers cannot be provided with a "two translators" solution, again, because all stubs must be tied to a concrete symbol which is visible at link time. An add-on RPC package can, therefore, at best make some form of "connection handle" first-class. Although anonymous, such a handle is not typed, and therefore introduces more possibility of error.

4 Implementation

To enhance portability, the core of the Concert/C compiler is implemented as a preprocessor which executes after the ANSI C preprocessor, and before the ANSI C compiler. The compiler shell `ccc` invokes the local ANSI compiler in preprocess-only mode, passes the output through the Concert/C preprocessor, and feeds the result to the local ANSI compiler.

Before producing an executable, `ccc` invokes a pre-linker, which scans the object files and performs operations which require a global view of the program. This usually produces a small amount of code and data which is turned into an object file and combined with the other files to form the final executable.

To support debugging with plain C tools, and also to ensure that error messages from the C compiler are understandable, `ccc` passes ANSI C code directly into its output stream wherever possible, examining it without modifying it. Line number information is captured and preserved. Unlike the *cfront* implementation of C++ [31], `ccc` does not "mangle" names. Concert/C programs, including ones instantiated by `create` can be debugged with tools such as `dbx` and `gdb`.

The compiler does, however, analyze all the declarations and statements in the program. Concert/C operations are checked and translated. The presence of any `port` function or `receiveport` declaration, or of an invocation of `create` (which may import one function or `receiveport` pointer), or a call to any function which has a function or `receiveport` pointer as an argument, triggers an analysis phase. The Concert/C type is examined and any missing attribute annotations are inferred and checked. These fully-annotated types are then fed into code which emits stub functions. Stubs are compiled for multiple protocols, with the final selection delayed until runtime.

The runtime is invoked on every RPC and every Concert/C operation. It dynamically selects among available protocols based on which pathways exist to the destination. Currently, it supports four protocols: SUN/RPC, OSF/DCE, our own UDP multicast protocol, and one based on shared memory "pointer passing" (between Concert/C processes implemented as threads in the same address space). The prototype permits Concert processes to interoperate with existing SUN/RPC components (such as NIS).

Many more details on the implementation are provided in [5].

5 Related Work

In the Concert project, we choose to extend multiple existing languages to best satisfy the conflicting goals of (1) hiding complexity and (2) building on an existing base of already-written code and programmer skills. A new language designed to support distributed computing (*e.g.*, Argus [20], SR [1], NIL [30], Emerald [10], Hermes [29]; a survey may be found in [7]) can hide complexity very well for those programmers willing to learn the new language. We will not engage in a feature by feature comparison with these languages, many of which have influenced Concert/C (particularly NIL and Hermes, which, like SR, share Concert/C's philosophy of providing both message-passing and RPC). Because such new languages must be learned, and programmers using them cannot easily draw on a large body of already-written code to perform routine tasks, they are not often used in the day-to-day practice of distributed computing. Also (although this is not inherent in the approach), the existing implementations of these new languages have not been "multi-protocol"; they are constructed in terms of a single protocol suite.

At the opposite extreme, commercial packages (such as OSF/DCE [25], Apollo NCS [19], PeerLogic's Pipes, Momentum's XIPC, Horizon's Message Express, or SUN ONC [32]) and software tools (such as Matchmaker [18], Courier [34], Horus [16], and HRPC [8]) permit programmers to continue using familiar

languages and to incorporate existing code. Many of these packages are "multi-protocol" at the transport level (they work over many transport protocols), although only HRPC is "multi-protocol" at the RPC protocol level (interoperating with other RPC-based tools, as does Concert/C).

All of the tools in this second category are either pure library based or (more usually) follow the two-translators model (with a separate stub compiler). As we have attempted to show in our previous discussion of the Concert/C language, this approach limits the extent to which complexity can be hidden from the programmer. In addition, few of these tools provide for process management, often they provide only RPC or only messaging, instead of both, and the RPC-based ones often support demultiplexing of requests at the server only indirectly via an associated thread package. Partly because function is defined at a low level, but *mostly* because of the limitations of the "two translators" approach, the library APIs of the most powerful tools in this class become quite complex.⁶ Tools attempting to address the residual complexity of library-based packages [24, 23] typically generate, in addition to stubs, a prologue to establish a program's initial connectivity, and also help automate the program construction process. Such tools are best at making simple "clients" that use a relatively static set of "servers." They provide only very limited help in writing servers, nor do they help when programs have evolving interrelationships.

The "single-translator" or "language extensions" approach of Concert/C is shared by some other efforts, most notably Linda [15]. Linda has a radically different model of computation than Concert/C (a shared tuple space), which makes it particularly effective for parallelizing single applications. However, we believe that Concert/C's model of computation, in which connectivity is modeled explicitly, is more appropriate for client/server applications (indeed, most of the tools cited above share Concert/C's view). In client/server applications, components often belong to semi-independent administrative domains, and the connectivity of the resulting "multi-application" is constantly changing.

Concurrent C [14] is also an extension to C, but we classify it as a new language because legacy C code almost always has to be reworked and must be recompiled in order to be incorporated into a Concurrent C program. Concurrent C has primitives for process creation and termination and has similar communication and synchronization semantics to Concert/C, but there are many differences reflecting Concurrent C's "single parallel application" design point. In Concurrent C, any process that has the other process' handle can invoke *any* of its exported procedures, and can also kill it; Concert/C provides a much finer granularity of control which is particularly important when crossing administrative boundaries. Concert/C provides all the necessary type system extensions to support the transmission of arbitrary C data structures across heterogeneous system boundaries; Concurrent C, lacking any such extension, requires either shared memory or a homogeneous distributed memory cluster. Concurrent C requires that the program bodies to be instantiated as processes be statically bound into a single program; Concert has no such requirement. One consequence of this difference is that Concert programs may be developed and instantiated by independent programmers in independent administrative domains and still communicate with each other.

PCN [12] is a new language with a C-compatible type system and the ability to link with C. However, when using PCN, all distributed logic is written strictly in PCN; C subroutines perform only local computation. With Concert/C, it is possible to make legacy C code directly invoke a remote function though originally written to perform only local function calls, or be invoked remotely though originally written to be invoked only locally. The degree of integration between old and new logic is therefore much greater with Concert/C.

As mentioned, while many tools support multiple transport protocols, almost none are designed to support multiple RPC protocols and also to interoperate with components using those protocols but not using the same programming tool. An exception is HRPC [8], a two-translators tool which does, however, have this multiple RPC protocols feature. Several things differentiate Concert/C from HRPC. Because it gets less help at translation time, HRPC uses a largely interpretive approach at runtime, in which explicit calls must be made through three layers of protocol support (control, data representation, and transport) in order to resolve the protocol. Concert/C pre-compiles for a set of target protocols at compile time and makes a single selection at runtime, after which all marshalling and demarshalling may be done by

⁶For example, OSF/DCE [25] has a published programming interface comprising 157 RPC and threads function calls, not counting the ones used privately by stubs.

compiled code. In HRPC, once a protocol is selected for a particular RPC handle, that protocol is fixed. In Concert/C, as a first-class pointer to a function or queue is passed around the network, it retains knowledge of *all* protocols capable of reaching the process which defines the function or queue. Different protocol selections may be made by different senders at different times in the lifetime of the binding.

6 Status

The current Concert/C prototype runs on AIX 3.2 on IBM Risc System/6000s, SunOS 4.1 and Solaris 2.2 on Sun workstations, and on OS/2 2.1. We have begun ports to VM/CMS and MVS, and are planning one to Windows/NT.

Programmers have used Concert/C to improve the performance of their applications in several ways, including parallelizing bottlenecks, and reducing communications by moving code next to its input data. For example, one large Concert/C application searches a database of genetic information on 64 Risc System/6000s in parallel [11]. Another Concert/C application uses similar search techniques to perform complex visual pattern recognition [27]. Concert/C has been used to connect a natural language textual query program to a collection of large, geographically dispersed text repositories [22]. The Global Desktop project uses Concert/C to enable the graphical interconnection of running applications, thus enabling cooperative computing over a network [21]. Yet another application has used Concert/C for the collection and analysis of time series data for the observation of seismic sensed phenomena [28].

The current prototype performs well. The round trip delay for an RPC that communicates a single integer takes 8 ms, little more than Sun RPC's 5 ms delay.⁷ Transmission of arrays achieves a bandwidth of 0.5 Mbyte/sec. This performance compares favorably with other systems for parallel programming, like pvm [33] and C-linda [15], as studied in [13]. The measurement program is in the Concert/C Tutorial [17]. Parsons [26] has compared the usability and performance of Concert/C with other tools for distributed programming including Isis [9] and PVM [33].

7 Future Work

Concert/C is the first of a set of compatible extensions to important programming languages. Thus, we may build Concert/Cobol, Concert/Fortran, Concert/C++, etc. Each of these languages will implement a distributed computing model called the *process model*: a distributed program is composed of a set of sequential processes communicating by RPC and asynchronous message passing [35, 2]. These languages will interoperate, so that a Concert/Cobol process could serve an RPC by a Concert/C process. To support data interoperability, IPC messages are mapped in and out of a Concert Universal type family, as described in [6].

We are expanding the support for OSF/DCE to further exploit DCE services such as naming and security, and to provide richer interoperability with DCE components not written using Concert/C. These improvements will be made available in the anonymous ftp version later this year. We are currently designing Concert/C++, and are investigating preliminary designs for Concert/Fortran. We are also interested in developing tools for process management and control, such as a distributed visualizer/debugger, and utilities which aid in the design of interoperable interfaces, using the internal function signature representation as an intermediate form to mediate the search for equivalent representations in different languages. We are also investigating improvements to the run-time to add group communication and fault-tolerant primitives such as causal and atomic multicast [9].

The Unix implementations of Concert/C, together with comprehensive documentation, including a tutorial [17], a programmer's manual [4] and example code, are available via anonymous ftp⁸.

⁷We measured this performance between IBM RS/6000 workstations connected by a 16Mbit token ring, and communicating via Sun RPC over TCP/IP.

⁸from `software.watson.ibm.com/pub/concert`

References

- [1] Gregory R. Andrews. Synchronizing Resources. *ACM Transactions on Programming Languages and Systems*, 3(4):405-430, October 1981.
- [2] J. S. Auerbach, D. F. Bacon, A. P. Goldberg, G. S. Goldszmidt, A. S. Gopal, M. T. Kennedy, A. R. Lowry, J. R. Russell, W. Silverman, R. E. Strom, D. M. Yellin, and S. A. Yemini. High-level language support for programming distributed systems. In *1992 International Conference on Computer Languages*, pages 320-330. IEEE Computer Society, April 1992.
- [3] Joshua Auerbach, Arthur P. Goldberg, German Goldszmidt, Ajei Gopal, Mark T. Kennedy, James R. Russell, and Shaula Yemini. Concert/C specification: Definition of a language for distributed C programming. Technical Report RC18994, IBM T. J. Watson Research Center, 1993.
- [4] Joshua Auerbach, Arthur P. Goldberg, German Goldszmidt, Ajei Gopal, Mark T. Kennedy, and James R. Russell. Concert/C manual: A programmer's guide to a language for distributed C programming. Technical report, IBM T. J. Watson Research Center, 1993. To be published, available from the authors.
- [5] Joshua S. Auerbach, Ajei S. Gopal, Mark T. Kennedy, and James R. Russell. Concert/C: Supporting distributed programming with language extensions and a portable multiprotocol runtime. Technical Report RC18995, IBM T. J. Watson Research Center, 1993.
- [6] Joshua S. Auerbach and James R. Russell. The Concert Signature Representation: IDL as intermediate language. Technical Report RC19229, IBM T. J. Watson Research Center, 1993. To appear the 1994 ACM SIGPLAN Workshop on Interface Definition Languages.
- [7] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3), September 1991.
- [8] B. N. Bershad, D. T. Ching, E. D. Lazowska, J. Sanislo, and M. Schwartz. Remote procedure call facility for interconnecting heterogeneous computer systems. *IEEE Transactions on Software Engineering*, 13(8):880-894, August 1987.
- [9] Kenneth P. Birman, Robert Cooper, et al. The ISIS system manual, version 2.0. Technical report, CS Department, Cornell, March 1990.
- [10] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65-76, January 1987.
- [11] A. Califano and I. Rigoutsos. FLASH: A Fast Look-Up Algorithm for String Homology. In *Proceedings First International Conference on Intelligent Systems for Molecular Biology*, Washington, DC, July 1993.
- [12] K. M. Chandy and S. Taylor. The composition of concurrent programs. In *Proceedings Supercomputing '89*. ACM, November 1989.
- [13] C. C. Douglas, Timothy G. Mattson, and Martin H. Schultz. A comparison of distributed and shared virtual memory systems on networks. Technical report, Department of Computer Science, Yale University, New Haven, 1993. YALEU/DCS/TR-975.
- [14] N. H. Gehani and W. D. Roome. *The Concurrent C Programming Language*. Silicon Press, 25 Beverly Road, Summit, NJ, 07901, 1989.
- [15] D. Gelernter and N. Carriero. Applications experience with LINDA. *SIGPLAN Notices*, 23(9):173-187, September 1988.
- [16] Phillip B. Gibbons. A stub generator for multilanguage RPC in heterogeneous environments. *IEEE Transactions on Software Engineering*, SE-13(1):77-87, January 1987.

- [17] Arthur P. Goldberg. Concert/C tutorial: An introduction to a language for distributed C programming. Technical Report RA218, IBM T. J. Watson Research Center, 1993.
- [18] Michael B. Jones and Richard F. Rashid. Mach and Matchmaker: Kernel and language support for object-oriented distributed systems. Technical Report CMU-CS-87-150, CS Department, CMU, September 1986.
- [19] Mike Kong, Terence H. Dineen, Paul J. Leach, Elizabeth A. Martin, Nathaniel W. Mishkin, Joseph N. Pato, and Geoffrey L. Wyant. *Network Computing System Reference Manual*. Prentice-Hall, Englewood Cliffs, NJ, 1990.
- [20] B. Liskov. Distributed programming in Argus. *Comm. ACM*, 31(3), March 1988.
- [21] Andy Lowry, Rob Strom, and Danny Yellin. The Global Desktop, IBM T. J. Watson Research Center. To be published, available from the authors.
- [22] Y.S. Maarek, D.M. Berry, and G.E. Kaiser. An information retrieval approach for automatically constructing software libraries. *Transactions on Software Engineering*, 17(8), August 1991.
- [23] Netwise. *C Language RPC TOOL*, 1989. Boulder, Colorado.
- [24] NobleNet. *EZ-RPC Manual*, 1992. Natick, Ma.
- [25] Open Software Foundation, Cambridge, Mass. *OSF DCE Release 1.0 Developer's Kit Documentation Set*, February 1991.
- [26] Ian Parsons. Evaluation of distributed communication systems. U. Alberta. Available from author.
- [27] I. Rigoutsos and R. Hummel. Distributed Bayesian Object Recognition. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition*, New York City, NY, June 1991.
- [28] Patricia Gomes Soares and Alan Randolph Karben. Implementing a Dlegation Model Design of an HPCC Application Using Concert/C. In *Proceedings of the 1993 IBM Centre for Advanced Studies Conference*, pages 729-738, Washington, DC, October 1993.
- [29] Robert E. Strom, David F. Bacon, Arthur Goldberg, Andy Lowry, Daniel Yellin, and Shaula Alexander Yemini. *Hermes: A Language for Distributed Computing*. Prentice Hall, January 1991.
- [30] Robert E. Strom and Shaula Alexander Yemini. NIL: An integrated language and system for distributed programming. In *SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, June 1983.
- [31] Bjarne Stroustrup. A history of C++: 1979-1991. *SIGPLAN Notices*, 28(3):271-297, March 1993.
- [32] Sun Microsystems. *SUN Network Programming*, 1988.
- [33] V. S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2(4):315-339, December 1990.
- [34] The Xerox Corporation. *Courier: The Remote Procedure Call Protocol*, December 1981. Technical Report XSI5 038112.
- [35] S. A. Yemini, G. Goldszmidt, A. Stoyenko, Y. Wei, and L. Beeck. Concert: A high-level-language approach to heterogeneous distributed systems. In *The Ninth International Conference on Distributed Computing Systems*, pages 162-171. IEEE Computer Society, June 1989.

Author Information

Joshua Auerbach is manager of Distributed Systems Software Technology at the IBM T J Watson Research Center, where he has been a Research Staff Member since 1983. His research has centered on problems of heterogeneity arising from system interconnection, including file systems, transport protocols, and now programming languages and RPC protocols. Email: jsa@watson.ibm.com

Arthur Goldberg is a technical assistant to the IBM Research Vice-President for Solutions, Applications and Services. He participated in the development of the distributed programming languages Concert/C and Hermes. He obtained his PhD in CS from UCLA in 91, and his AB in Astrophysics from Harvard in 77. Email: artg@watson.ibm.com

German Goldszmidt is a final-year PhD student in the Department of Computer Science at Columbia University. He has worked full-time and part-time for IBM Research for the past five years. His thesis research is in the area of network management. Email: german@cs.columbia.edu

Ajei Gopal received his PhD in Computer Science from Cornell University. He is a Research Staff Member at IBM Research, and manager of the Cluster Systems group. He previously worked in the Distributed Systems Software Technology group. His research interests center on parallel and distributed systems and fault-tolerance. Email: ajei@watson.ibm.com

Mark Kennedy received a BA in Chemical Physics and an MS in Computer Science from Columbia University. He has been a member of the Distributed Systems Software Technology group at IBM Research for the last four years. His research interests include distributed systems and doing anything to make it easier to program large collections of Unix-based workstations. He was a co-winner of the Winter '89 Usenix Humor Contest. Email: mtk@watson.ibm.com

Josyula R. Rao received his PhD in Computer Science from the University of Texas at Austin in 1992. He is currently a Research Staff Member at IBM Research in the Distributed Systems Software Technology group. His research interests include distributed systems and formal methods. Email: jrrao@watson.ibm.com

James Russell received his PhD in Computer Science from Cornell University. He is a Research Staff Member at IBM Research, where he has been working in the Distributed Systems Software Technology group for the past three years. His research interests focus on the design and implementation of programming languages for distributed applications. Email: jrussell@watson.ibm.com

Evolving Mach 3.0 to a Migrating Thread Model

Bryan Ford Jay Lepreau

University of Utah

Abstract

We have modified Mach 3.0 to treat cross-domain remote procedure call (RPC) as a single entity, instead of a sequence of message passing operations. With RPC thus elevated, we improved the transfer of control during RPC by changing the thread model. Like most operating systems, Mach views threads as statically associated with a single task, with two threads involved in an RPC. An alternate model is that of migrating threads, in which, during RPC, a single thread abstraction moves between tasks with the logical flow of control, and “server” code is passively executed. We have compatibly replaced Mach’s static threads with migrating threads, in an attempt to isolate this aspect of operating system design and implementation. The key element of our design is a decoupling of the thread abstraction into the *execution context* and the *schedulable thread of control*, consisting of a chain of contexts. A key element of our implementation is that threads are now “based” in the kernel, and temporarily make excursions into tasks via upcalls. The new system provides more precisely defined semantics for thread manipulation and additional control operations, allows scheduling and accounting attributes to follow threads, simplifies kernel code, and improves RPC performance. We have retained the old thread and IPC interfaces for backwards compatibility, with no changes required to existing client programs and only a minimal change to servers, as demonstrated by a functional Unix single server and clients. The logical complexity along the critical RPC path has been reduced by a factor of nine. Local RPC, doing normal marshaling, has sped up by factors of 1.7–3.4. We conclude that a migrating-thread model is superior to a static model, that kernel-visible RPC is a prerequisite for this improvement, and that it is feasible to improve existing operating systems in this manner.¹

1 Introduction and Overview

We begin by defining and explaining four concepts that are key to this paper. They are kernel and user threads, remote procedure call, static threads, and migrating threads. We explain how kernel threads interact in implementing RPC, and the difference between implementing RPC with static and migrating threads.

Threads As the term is used in most operating systems and thread packages, conceptually a *thread* is a *sequential flow of control*[4]. In traditional Unix, a single process contains only a single kernel-provided thread. Mach and many other modern operating systems support multiple threads per process (per *task* in Mach terminology), called *kernel threads*. They are distinguished from *user threads*, provided by user-level thread packages, which implement multiple threads of control atop kernel-provided threads, by manipulation of the program counter and stack from user-space. In the rest of this paper, we use the term “thread” to refer to a kernel thread, unless qualified.

In most operating systems, a thread includes much more than the flow of control. For example, in Mach 3.0[24] a thread also (i) is the *schedulable entity*, with priority and scheduling policy attributes; (ii) contains *resource accounting statistics* such as accumulated CPU time; (iii) contains the *execution context* of a computation—the state of the registers, program counter, stack pointer, and references to the containing task and designated exception handler; (iv) provides the *point of thread control*, visible to user programs through a *thread control port*.²

¹This research was sponsored in part by the Hewlett-Packard Research Grants Program and by the OSF Research Institute.

²In Mach, a *port* is a kernel entity that is a capability, a communication channel, and a name. If one has the name of a port,

RPC *Remote procedure call*, as the name suggests, models the procedure call abstraction, but is implemented between different tasks. The flow of control is temporarily moved to another location (the “procedure” being called) and later returned to the original point and continued. RPC can be used between *remote* computing nodes, but is often used between tasks on the same node: *local RPC*. This paper focuses only on the local case; in the rest of the paper, for brevity, we use the unqualified term “RPC” to refer only to local RPC (more restricted than the usual use of the term). When a thread in a *client* task needs service provided by another task, such as opening a file, it bundles up a packet of data containing everything the service provider (the *server*, or file system in this case) needs to process the request. This is known as *marshaling a message*. The client thread then invokes the kernel to copy the message into the server’s address space and allow the server to start processing it. Some time later, the server returns its results to the client in another message, allowing the client to continue processing.

It is important to note that although virtually all modern operating systems provide an RPC abstraction at some level, there is a continuum in that support. Some OS’s support RPC as a fully kernel-visible entity (Amoeba[29]), some provide a kernel interface and special optimizations for the combined message send and receive involved in RPC, but fundamentally understand only one-way message passing (Mach), while some support RPC only through libraries layered on other inter-process communication (IPC) facilities (Unix).

Static Threads The difference between static and migrating threads lies in the way the control transfer between client processing and server processing is implemented. In RPC based on static threads, two entirely separate threads are involved: one confined to (or *static* in) the client task, and the other confined to the server task. When the client invokes the kernel to start an RPC, the kernel not only copies the client’s message into the server, but also puts the client’s thread to sleep and wakes up a thread in the server to process the message. This thread, known as a *service thread*, was created previously by the server for the sole purpose of waiting around for RPC requests and processing them. When the service thread is finished with the request, it invokes the kernel, which puts the server thread back to sleep and wakes up the client thread again. In switching control from one thread to another, a full *context switch* is involved—a change of address mappings, task, thread, stack, registers, priority, etc., including an invocation of the kernel scheduler.³

With RPC based on static threads, the server’s *computational resources* (the right to use the CPU) are used to provide service to the client. In the object-oriented world, this is known as an “active object” model[9], because a server “object” contains threads that actively provide service.

Migrating Threads If RPC is fully visible to the kernel, an alternate model of control transfer can be implemented. *Migrating threads* allows threads to “move” from one task to another as part of their normal functioning. In this model, during an RPC the kernel does not block the client thread upon its IPC kernel call, but instead arranges for it to continue executing in the server’s code. No service thread needs to be awakened by the kernel—instead, for the purposes of RPC, the server is merely a passive repository for code to be executed by client threads. It is for this reason that in the object-based world, this is called the “passive object” model. Only a partial context switch is involved—the kernel switches address space and some subset of the CPU registers such as the user stack pointer, but does not switch threads or priorities, and never involves the scheduler. There is no server thread state (registers, stack) to restore. The client’s own computational resources (rights to the CPU) are used to provide services to itself. Note that *binding* in this model can be very similar to that in a thread switching model and will be detailed in Section 6.2.

Although most operating systems support RPC using the static thread model, whether kernel-visible or not, it is important to note that this is not the case for *all* system services. All systems using the “process model”[14]⁴ for execution of their own kernel code (e.g., Unix, Mach, Chorus, Amoeba), actually “migrate” the user’s thread into the kernel address space during a kernel call. No context switch takes place—only the stack and privilege level are changed—and the user thread’s resources are used to provide services to itself.

one can perform operations on the object it represents.

³Sometimes this invocation of the scheduler is heavily optimized and inlined into the IPC path, but it is still there.

⁴The “process model” is in contrast to the “interrupt model,” as exemplified by the V operating system[8], in which kernel code must explicitly save state before potentially blocking.

Static vs. Migrating Threads In actuality, there is a continuum between these two models. For example, in some systems such as QNX[20], certain client thread attributes, such as priority, can be passed along to (“inherited by”) the server’s thread. Or a service thread may retain no state between client invocations, only providing resources for execution, as in the Peregrine RPC system[21]. Thus it can become impossible to precisely classify every thread and RPC implementation. However, most systems clearly lie towards one end of the spectrum or the other.

1.1 Providing Migrating Threads on Mach

Mach uses a static thread model, and a thread contains all of the attributes outlined in the “Threads” paragraph above. In our work, we decoupled these semantic aspects of the thread abstraction into two groups, and added a new abstraction, the *activation stack*, which records the client-server relationships resulting from RPCs. A *thread* is now only: (i) the logical flow of control, represented by a stack of *activations* in tasks; and (ii) the schedulable entity, with priority and resource accounting attributes. An *activation* represents: (i) the execution context of a computation, including the task whose code it is executing, its exception handler, program counter, registers, and stack pointer; and (ii) the user-visible point of control.

The abstraction exported to user code that corresponds to the old “thread” abstraction is now what we internally call the “activation.” This is not only what makes sense for the needs of user programs, but also provides compatibility with original Mach 3.0.

The real thread as defined above, the schedulable entity, is no longer subordinate to a task. By making RPC a single identifiable entity to the kernel, and explicitly recording the relationship between individual activations in the activation stack (which result from RPC), we have elevated RPC to an entity fully visible to, and supported by, the kernel, instead of a sequence of message passing operations. Our thread abstraction now more closely models the original *conceptual* basis of a thread: a logical flow of control. It turns out that elevating the thread and RPC abstractions also enhances *controllability*, because the kernel can now take more elaborate and precise actions on a single activation or on the entire thread. For example, it can propagate information (“alerts”) along the chain of activations. Another benefit of introducing to the kernel the notion of an inter-task RPC, is that a number of aggressive IPC optimizations become possible. This was one of our original motivations, but many other benefits have since surfaced.

1.2 Outline of Goals and Benefits

Our original goals in this project were several: (i) change Mach 3.0’s thread model to a migrating one, (ii) retain backwards compatibility, and (iii) enable performance improvements via RPC optimizations not possible with static threads. During the design and implementation, we discovered we could achieve much more: (iv) ordinary message-based (fully marshaled) RPC became much faster, (v) thread controllability was enhanced, (vi) kernel code became much simpler, (vii) an apples-apples comparison of static vs. migrating threads was achieved, and (viii) several other advantages, discussed below, became evident.

In the rest of this paper we describe this work in detail. We first discuss related work, then cover the advantages of a migrating thread model, describe our kernel implementation and interface, including discussion of the thread controllability issues, examine how RPC works in the new system, and mention how the Unix server could be changed to better leverage migrating threads. Finally, we present the implementation status and preliminary results, outline future work, and the conclusions we draw from this work.

2 Related Work

Most operating systems use a static thread model, but there are a number of exceptions. Sun’s Spring[19] operating system supports a migrating thread model very similar to ours, although it uses different terminology. Spring’s “shuttle” corresponds to our “thread,” and their “thread” corresponds to our “activation.” Spring addresses the controllability issues but did not have to be concerned with backwards compatibility. Alpha[11] was probably the first system to fully adopt migrating threads. It is oriented to real-time constraints, and its migrating thread abstraction is especially important for carrying along scheduling, exception-handling, and resource attributes. In both of these systems a thread can migrate across nodes in a distributed environment, and indeed Alpha’s term for a migrating thread is a “distributed thread.” Psyche[27] is a single-address-space system that supports migrating threads. The Lightweight RPC system[3] on Taos exploited

migrating threads (control transfer) as a critical part of its design, but focused on high-performance local RPC, and included additional data transfer optimizations. This makes it difficult to isolate the benefits of the improved control transfer. Object-oriented systems have traditionally distinguished between “active” and “passive” objects, corresponding to static and migrating thread models[9]. Clouds[16] exemplifies a passive object (migrating thread) model, while Emerald[5], as we do, provides both active and passive objects—support for both styles of execution. Chorus[26] can use only thread-switching between user-level tasks, but between tasks running in the kernel’s protection domain it has “message handlers” which operate in a migrating thread model.

We believe that many of these migrating threads systems did not fully address the attendant controllability issues—did not fully support debugging or need to provide Unix signal semantics, for example.

“Scheduler activations”[1] are kernel threads with special support for user-level scheduling. Scheduler activations are concerned primarily with the behavior of kernel threads *within* a protection domain, while our work deals with thread behavior *across* protection domains. As such, these works are largely orthogonal and theoretically could be combined in the same system, but we do not deal with this issue here.

Except for LRPC on Taos, all of the existing systems supporting migrating threads were designed this way from the start. They are all different from traditional operating systems in many ways other than thread model. To our knowledge, heretofore the thread model issue itself has not been separated out and examined, comparing all of performance, functionality, and simplification. Our goal is to do this by comparing the two thread models in the same operating system, providing information focused on the thread model. By implementing migrating threads on Mach 3.0, we also demonstrate how an existing operating system with static threads can be adapted to migrating threads.

3 Motivation

A migrating thread approach has several advantages which are outlined in this section. The majority of the benefits are linked to use with RPC and are described first. But there are also controllability advantages for threads during *all* kernel interaction, and these are outlined in section 3.2. In the context of the Alpha OS, [11] also discusses many advantages offered by migrating threads.

3.1 Remote Procedure Call

Many of the advantages of migrating threads stem from their use in conjunction with RPC. Migrating threads provide a more appropriate underlying abstraction on which to build RPC interfaces than do static threads. Many of the problems with static threads stem from the semantic gap between the control model—a procedure call abstraction within a single thread of control, and the mechanism used to implement the model—two threads executing in different tasks. Using migrating threads for RPC provides benefits in performance, functionality, and in ease of implementation. Since RPC is very frequently used[3], especially in newer microkernel-based operating systems where most internal system interactions are based on RPC, this aspect of the system can be of great importance in determining the performance and functionality of the system as a whole.

Invocation Efficiency

For RPCs to be performed in the static thread model, two threads, one in each task, must synchronize in the kernel. Two thread-to-thread context switches are required during the operation: one on call and one on return. However, in the migrating thread model, the entire RPC can be performed by just one thread that temporarily moves into the server task, performs the requested operation, and then returns to the client task with the results. No synchronization, rescheduling, or full context switch need be done.

Thread migration also permits optimizations such as those done in LRPC[3] and in other flexibly structured or shared address space systems, e.g., Lipto[15], FLEX[7], and Mach In-Kernel Servers[22, 17]. In these systems there is some degree of inter-domain memory sharing or protection relaxation, thus blurring domain boundaries. RPC implemented by threads that migrate from one domain to another can take advantage of this boundary blurring, providing many optimizations in argument passing and stack handling. At the limit, RPC implemented in a migrating thread model could be specialized to a simple procedure call.

These advantages apply in general to any *service invocation* mechanism, not just large servers invoked through RPC. For example, in an object-based environment, invocation of relatively fine-grained objects is prohibitively inefficient if all objects must be active. With passive objects, it is more feasible to apply similar invocation abstractions to both medium and course-grained objects.

While the existence of fast, efficient microkernels based on static threads demonstrates that high performance is possible in that model, such systems often impose semantic restrictions that distort their implementation towards a migrating thread model. For example, QNX[20], a commercial real-time operating system, supports only unqueued, synchronous, direct process-to-process message passing with priority inheritance; this design makes it a *de facto* migrating threads system. Other microkernels, such as L3[23], retain the full semantics of static threads, but to achieve high performance must impose severe restrictions on the flexibility of scheduling and other aspects of the system not directly related to RPC.

Thread Attributes and Real-time Service

In the static thread model, when a client task performs an RPC, control is transferred to an entirely different thread that has its own scheduling parameters such as execution priority, as well as other attributes such as resource limits. Unless specific actions are taken, the attributes of the thread in the server will be completely unrelated to those of the client thread. This can cause the classic problems of *starvation* and *priority inversion*[13], when a high-priority client is unfairly made to compete with low-priority clients that are accessing the same server. On the other hand, if the client thread migrates into the server to perform the operation, all such attributes can be properly maintained with no extra effort. Obviously, this issue is of particular importance to systems providing real-time service.

A related advantage is in resource accounting, which can be made more accurate since the work done in a server on behalf of a client can automatically be so attributed.

Interruptions during RPC

Often, due to asynchronous conditions, it is desired to interrupt an RPC in which a client is blocked, either temporarily or permanently. To do this cleanly in the static thread model, it is not enough merely to abort the message send/receive operation, because the server will continue processing the request without any indication that the client no longer desires its completion. If some entity wants to abort an RPC in which a thread is blocked, it must find the server to which the RPC is directed, know how to interact with that server enough to send it a request to abort an RPC operation, and provide the server with some kind of identification specifying which RPC is to be aborted. This usually proves to be a complex and difficult process. In addition, every server that may be accessed must support these abort operations. This can be difficult to guarantee in practice, especially if any user-mode task can set itself up as a "server" and allow other user threads to make RPCs to it, as Mach 3.0 allows. Migrating threads, on the other hand, provide a channel through which standardized requests for interruption can be propagated.

Server Simplification

In the case of "personality servers" that emulate monolithic operating systems such as Unix or OS/2, we expect migrating threads to simplify the server, because the original operating system on which the server is based is likely to have used a limited migrating thread model, in which threads "migrate" into the monolithic kernel for system calls. Maintaining this model in the personality server should achieve greater code re-use and simplify the handling of system call interruptions, thread management, and control mechanisms such as Unix signals.

We also expect migrating threads to simplify RPC service in servers, due to the anticipated simpler management of activation pools than thread pools, as described in Section 6.2.

Kernel RPC Path Simplification

As later shown by our results in Section 8.5, migrating threads greatly simplify the kernel RPC path as well. RPC paths based on migrating threads tend to be short and flow naturally, while optimized RPC paths based on static threads are often long, convoluted, and contain innumerable tests.

3.2 Thread Controllability and Kernel Simplification

A migrating threads implementation gives other controllability and simplification benefits unrelated to RPC. In a static thread model, threads are often intended to be *completely controllable resources*. Ideally, in this model, any entity with appropriate privilege, such as a program holding a “thread control port” in Mach 3.0, is able arbitrarily to stop a thread and modify its state, at any time. Conceptually, threads execute only user-mode instructions, and therefore there is never a time when system integrity could be violated by manipulation of the thread.

Unfortunately, this model in its purest form does not work in real operating systems. Threads must be able to invoke kernel-level code in order to communicate with other entities in the system, if they are to do anything more than pure computation. Since a thread executing unknown kernel code may *not* be arbitrarily manipulated, the model of complete controllability must break down somewhat: it must be possible to defer or reject thread control operations when necessary.

Traditional operating systems have various ways of working around this problem which usually work, but are often complex, inconsistent, and unpredictable. For example, Mach 3.0 provides a thread control operation which aborts a system call in which the target thread is blocked, so that the thread can be manipulated. However, many kernel operations cannot be aborted in a transparent, restartable way, so the entity trying to control the thread may have to wait an arbitrary length of time, or retry an arbitrary number of times, before it can safely do so. If this is the case, who is really being controlled—the target thread, or the thread trying to control it?

Since the complete controllability model is not realistic anyway, reducing the ambitiousness of the model, to allow for migrating threads, provides more precisely defined semantics for thread manipulation. In fact, by forcing the boundaries of controllability to be explicitly defined, and recording the flow of control across tasks, additional thread control mechanisms such as cross-domain “alerts” can be provided by the kernel. Defining the boundaries of control also makes all control mechanisms much simpler to implement, as shown in Section 8.5.

4 Kernel Implementation

In this section we describe the underlying structure of our implementation of migrating threads in the Mach 3.0 microkernel. Many of the techniques we used could be similarly applied to other traditional multithreaded operating systems such as monolithic Unix kernels.

4.1 Thread Implementation

Conceptually, a traditional Mach 3.0 user thread started executing in a particular task, and occasionally trapped into the kernel to communicate with “outside” entities. The kernel later returned from the system call and resumed the user code. The initial and normal location of a thread was in user space, and threads only “visited” the kernel occasionally, to request services.

In our migrating thread implementation, the situation is in a sense reversed. A thread starts executing as a purely kernel-mode entity, and later makes an upcall[10] into user space to run user code. Conceptually, the kernel is “home base” for all threads: the only time user-level code is executed is during “temporary excursions” into a task. A thread executing in user mode is associated with the task in which it is currently running, but a thread running in the kernel is not tightly associated with *any* user-level task.

While a thread in the kernel can now make upcalls into user space, the traditional kernel/user interface is still preserved. Once a thread is executing in user space, it can make calls *back* to the kernel in the form of traps and exceptions. Alternatively, the kernel can make further upcalls into the same or a different user task. This redefinition of the kernel/user interface is the primary mechanism supporting migrating threads in our implementation.

A distinction should be made between the “kernel” and what we refer to as “glue” code. The kernel is conceptually a protection domain much like a user-level task, in which threads can execute, wait, migrate in and out, and so on; its primary distinction is that it is specially privileged and provides basic system control services. Glue code is the low-level, highly system-dependent code that enacts the *transitions* between all protection domains, both user and kernel. The distinction between the kernel and glue code is often

overlooked because both types of code usually execute in supervisor mode and are often linked together in a single binary image. However, this does not necessarily have to be the case; for example, in QNX[20], the 7K “microkernel” consists of essentially nothing but glue code, while the “kernel proper” is placed in a specially privileged but otherwise ordinary process. It will become clear in later sections that even though the kernel and glue code may still be lumped together, in the presence of migrating threads the distinction between them becomes extremely important.

4.2 Control Abstractions and Mechanism

Even in the static thread model, in practice the goal of complete controllability of threads cannot be fully realized. While the case of a thread being in the kernel can to some extent be worked around as a special case, with the addition of migrating threads the controllability issue must be more carefully considered. Now, not only is the kernel “out of bounds” for thread control, but in order to maintain protection between tasks, threads that have migrated to other protection domains may also be uncontrollable. For example, if one thread in a client migrates into a server for an RPC, it would not be permissible for another thread in the same client to stop or manipulate the CPU state of the first thread while executing server code.

To provide controllability and protection at the same time, we split the concept of a “thread” into two parts: the part used by the scheduler, and the part providing explicit control. The first, which we still refer to as the “thread,” migrates between tasks and enters and leaves the kernel. The second, a user-mode invocation or *activation*, remains permanently fixed to a particular task. Arbitrary control is permitted *only on a specific activation*, not on the thread as a whole.

Whenever a thread migrates into a task (including the initial upcall from the kernel on thread creation), an activation is added to the top of the thread’s “activation stack.” When a thread returns from a migration, the corresponding activation is popped off the activation stack. This new kernel-visible abstraction, the stack or chain of activations, helps provide controllability.

Activations are created either implicitly during thread creation, or explicitly by servers expecting to receive incoming migrating threads. An explicitly created activation is *unoccupied* until a thread migrates into the task and “activates” it.

Within the kernel, control of activations is implemented primarily with *asynchronous procedure calls*, or APCs, similar to asynchronous traps (ASTs) in monolithic kernels. When returning from the kernel into an activation, glue code checks for APCs attached to the activation and if present, calls them. For example, to suspend an activation, an APC is attached to that activation which will block until resumed. Previously, Mach dealt with thread suspension as part of the scheduler, adding more complexity to its already-complex state machine; now the scheduler knows nothing about one thread suspending another. Instead, the kernel’s ordinary blocking mechanism is used, in which a thread only “suspends” itself.

4.3 Kernel Stack Management

Since the activation chain can be broken at any point, all linkage information between activations is stored in the activations themselves, and a single kernel stack is sufficient for the entire thread. This is also required for it to be possible to do task migration across nodes in a distributed system, because state held on a kernel stack cannot be easily encapsulated for transport. This explicit saving of state is generically known as using a *continuation*, although our implementation is very different from the way continuations have been used in Mach in the past[14]. In particular, we confine continuations purely to “glue” (transition) code; all high-level kernel code uses an ordinary process model.

5 Controllability: Semantics, Interface, and Implementation

In this section we describe the semantics of thread control operations, the interface to those operations, and some aspects of the implementation. We believe our approach could be similarly applied to other traditional multithreaded operating systems.

5.1 Thread Control Interface

In the original Mach kernel, threads were exported to user-mode programs in the form of *thread control ports*, through which control operations could be invoked. In our system, while threads still exist, the control

abstraction presented to user-level code is instead the *activation control port*. This can work because the old thread execution abstraction exported to the user was bound to a single task, like activations are now. We maintain compatibility with existing Mach code by making activation control ports direct replacements for thread ports at the binary level—all system calls which previously expected or returned thread ports now use activation ports instead. For compatibility at the source level, appropriate synonyms are provided.

Alerts

In our migrating threads implementation, we have provided the functionality of Mach 3.0's `thread_abort` call, which aborts an in-progress kernel operation and returns control to user code. However, we have provided it in a cleaner and more general form. An *alert* is a form of asynchronous message passed from a client to the kernel or a server it is calling, asking the callee to abort the requested operation and return control to the client as soon as possible. Alerts are primarily an information-passing mechanism supported by the kernel in a uniform way. They do not in themselves provide control over threads, because they have no forcefulness: alerts are merely "requests," not "demands." We currently implement only a polling interface for a server to discover an alert, but probably will also provide an exception interface in the future. Alerts are much like those in Spring[19], The "Alert" and "TestAlert" facilities of Taos[4] are analogous, but apparently do not operate cross-domain.

By default, new activations added to a thread's stack have alerts blocked, to prevent interference with an unwary server's functioning. The kernel is already capable of honoring most alerts, and new servers written to work with migrating threads can be designed to honor them too. In effect, we have provided a generic interruption request mechanism which works uniformly for both migrating RPCs and kernel calls.

We also provide another operation which first generates an alert at the target activation, then breaks the chain, returning control to the client immediately. This works much like termination, discussed below.

Suspension

In Mach 3.0 thread semantics, the basic purpose of suspending a thread is to *prevent it from executing any more user-mode instructions* until it is resumed. Therefore, suspending a task's threads turns that task into a "passive entity," allowing its address space and other state to be examined or modified without interference from its threads. It is not required that all of the thread's computation be immediately stopped, as long as that computation does not implicitly reference the thread's task. For example, explicit device I/O could be allowed to proceed while the thread is suspended, but kernel `copyin` and `copyout` operations, which implicitly affect the task's address space, could not.

Our current implementation allows such kernel activity to proceed. We do not expect this to be a problem in practice, but in any case it should be solved as a side-effect of related work. In that work we are further separating kernel code from "glue" code (described earlier). When an activation is suspended, the kernel ensures that neither user-mode or glue instructions will be executed *in that activation*. If the thread is executing elsewhere, it will not be affected until it attempts to return to the suspended activation.

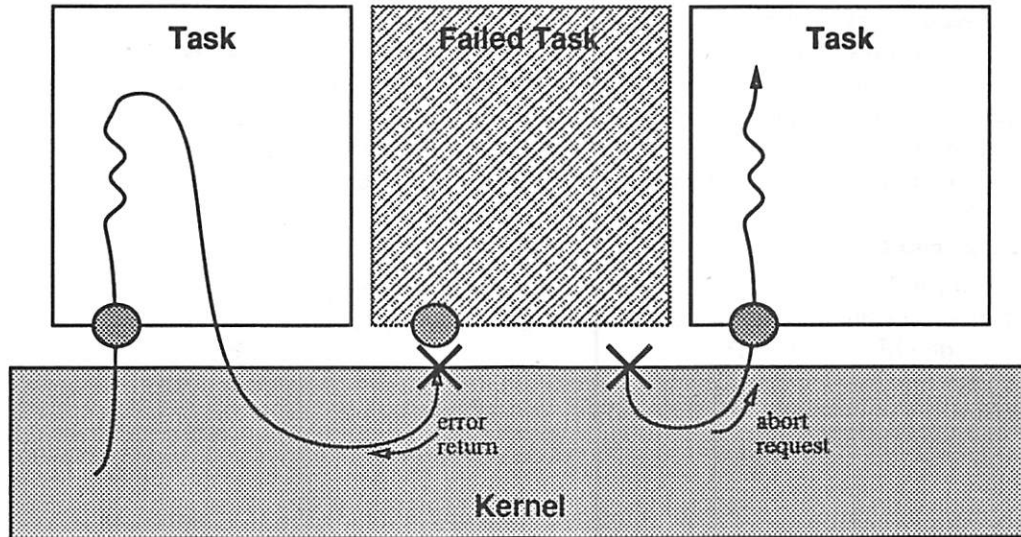
To maintain correct suspension semantics in this model, implicit references made by kernel system calls to the caller's task must be confined to glue code. This is relatively easy to do in Mach because most kernel calls are implemented as generic RPCs to kernel objects: only the low-level RPC code needs to obey the control semantics, not the actual code implementing the kernel calls.⁵

Termination

The termination mechanism in our implementation is illustrated in Figure 1. If an activation not at the top of a thread's activation stack is terminated, or if the thread is in a kernel call at the time, then the thread splits into two separate threads with identical scheduling parameters. One thread is left with the top part of the activation stack, above the terminated activation, and the other thread is given the bottom part. The thread with the upper segment continues executing in the topmost server uninterrupted, ensuring that thread termination does not violate protection. An alert is automatically propagated upward through

⁵Note that traditional Unix suspension semantics are stronger than Mach suspension semantics: they imply that kernel operations are actually completed or aborted before the thread is suspended. On Mach, both with static and migrating threads, implementing Unix suspension semantics involves other Mach control operations in addition to simple Mach suspension.

Figure 1: Activation Termination



this thread, providing a hint that the work being done is probably no longer of value. The thread given the bottom part of the activation stack returns to its now-topmost activation with an appropriate error code. This is essentially the same as the termination mechanism used in Spring.⁶

In our system, not only can a thread be split when running in user mode in a more recent activation than the terminated activation, but also when the thread is executing in the kernel, on behalf of the terminated activation. Previously, attempting to terminate a thread could potentially block for some time, while the caller waited for the victim to leave the kernel or otherwise get to a “clean point.” In the new model, terminating an activation is *always* an immediate operation: if the terminated activation happens to be calling the kernel, then it is left behind to finish whatever operation it was performing and quietly self-destruct afterwards. The issue of glue and kernel operations implicitly affecting the thread’s task is dealt with as described above under “Suspension.”

Our termination mechanism required careful planning of kernel data structures and locking mechanisms; in particular, the line between the kernel and glue code had to be defined precisely. Once worked out, however, this technique not only added additional controllability, but considerably *simplified* the implementation of control mechanisms in the kernel, as we show in Section 8.5.

CPU State

The original Mach 3.0 design provided thread operations which, in the “ideal” complete controllability model, would allow a thread’s entire CPU state to be saved, restored, examined, and modified at any time. All CPU state operations were provided by two primitives, `thread_get_state` and `thread_set_state`, defined to produce consistent results only while the target thread was suspended. However, because of the problems with the complete controllability model, many of the things for which the CPU state control mechanisms are commonly thought to be useful, in fact cannot be reliably implemented in bounded time in Mach 3.0[18]. For example, encapsulation of a task’s state for checkpointing or transportation to another node either may require the controlling thread to wait an arbitrarily long time, or else requires aborting kernel operations, yielding potentially inaccurate state.

Since the existing CPU state control operations are already problematic, and it would be difficult to achieve complete backward compatibility with them, we chose to structure these operations in our migrating threads implementation to fit *current uses* of these operations: in particular, those made by the Unix server

⁶We are considering providing alternate termination semantics which fully preserve the synchronous nature of RPC. The terminated activation would be merely spliced out, so control is returned to the earlier activations only after those later in the chain have exited. It may not be possible to guarantee these semantics over a partitionable network, however.

and emulator, and by application that create their own threads and control them in straightforward ways.

Mach 3.0 requires `thread_abort` to be called on a thread just before examining or setting its state, unless the thread has just been created. Otherwise, the state operation could work with “stale” information, producing useless results. Under migrating threads, aborting an activation before manipulating its state is not strictly required. If not done, the CPU state operations wait patiently until the thread is in the target activation, without interfering with its functioning. Thus we have loosened the restrictions on these operations while maintaining backwards compatibility with their only valid usage in original Mach 3.0.

Scheduling Parameters

The final Mach 3.0 thread control operations that must be mapped to activation operations are those managing thread scheduling parameters such as priority, scheduling policy, and CPU usage statistics. Unlike the operations described above, these operations are still conceptually performed on threads rather than activations. However, the original Mach 3.0 thread control ports have become activation ports, raising the question of how the interface for these operations should be handled.

Since every active activation is attached to exactly one thread, in our current implementation we export thread operations as operations on activations, and, in the kernel, redirect the operations to the attached thread. However, this raises a protection problem, since *any* activation in the thread can modify the global scheduling state. For example, a server could lower a thread’s maximum priority (which cannot be raised without special privileges) while processing an RPC, leaving the client with a “crippled” thread upon return. In our initial implementation, this is not a problem in practice because the Unix server is trusted by all clients. A better solution would be to provide an activation operation which forbids future activations higher in the stack from changing global thread state. Thread state could still be manipulated from that activation or lower (assuming it has not also been forbidden at a lower level).

5.2 Task Control Interface

Most task control operations work the same way under migrating threads as in the original Mach design. Others were modified in straightforward ways to match the new thread model. In particular, the `task_threads` call now returns a list of the activation ports of the task, instead of thread ports. When a task is suspended, resumed, or terminated, all of the activations within it (instead of threads) are similarly suspended, resumed, or terminated.

6 Migrating RPC

Once the basic kernel mechanism for supporting migrating threads was in place, it remained to demonstrate its effect on RPC performance and complexity. Because our focus at this point is primarily on control transfer during RPC, our initial implementation retains the original Mach data transfer interface, based on marshaling and unmarshaling done by user-mode stubs. In this section we describe this RPC system, as well as the changes required to servers to make them support migrating RPC. (No changes were required to make them run with traditional RPC, since the kernel itself is almost completely backward compatible.)

6.1 Client-side

From the client’s point of view, RPC semantics, including binding, are unmodified. Existing binaries with normal `mach_msg` calls are supported: the kernel checks the message options to make sure they specify a true RPC, and checks the destination port to ensure that the server is capable of handling migrating RPCs. In practice, almost all MIG-generated `mach_msg` calls meet these requirements, so most clients automatically make use of migrating RPC. Note that the data can still contain port rights and out-of-line memory.

6.2 Server-side

Initializing a server to support migrating RPCs is done in nearly the same way as in servers supporting only thread-switching RPC. In accomplishing the major portion of binding, the server exports send rights to clients, exactly as before. In addition, the server must create one or more unoccupied activations, each containing a pointer to a stack in its own address space, and in the final portion of binding, the entry point

of its normal dispatch function.⁷ Providing this information to the kernel can be encapsulated within a function, e.g., in the `cthreads` package.

Traditional static-thread RPC is still supported automatically. A large pool of server threads is no longer needed, but at least one must still exist to process occasional asynchronous messages, because in the current implementation this is used as a fallback mechanism when migrating RPC cannot be used.

When a migrating RPC is made into the server, the kernel allocates an unoccupied activation from the server's pool, copies the incoming message onto the server stack, and makes an upcall into the server task to the dispatch routine. This MIG-generated routine is identical to the one used to dispatch traditional messages, except that it returns through a special kernel entrypoint. On return, the kernel does not need to do any security checks or port manipulation, and the reply port provided by the client in the `mach_msg` call is never used at all.

If a migrating RPC is attempted and the kernel discovers that there are no activations currently available, in our initial implementation the kernel falls back to the normal message path, causing a normal message to be queued to the port. This is not ideal, and we plan to detect when the last available activation is about to be used for a migrating RPC, and instead of immediately making the requested RPC, temporarily "sidetrack" and make a special notification upcall into the server. At this point the server can create more activations if it deems this desirable. If it does, it returns them to the kernel and the original RPC can proceed. Otherwise, it returns immediately and the RPC blocks until a stack is freed.

When this is implemented, server management of an activation pool should be substantially more straightforward than management of a thread pool, for several reasons. The resources will be allocated on demand, by client threads themselves, instead of resource needs having to be predicted in advance, by the server. The server can use some simple decay function to deallocate activations (which are cheap for the kernel to manage since they are simply passive data structures, in contrast to kernel threads). In contrast, with a thread pool, a server is separated by the kernel "wall" from clients' requests—if inadequate numbers of server threads are present, client messages build up in the server's queues without its knowledge. The server has a more complex job as it attempts to keep the number of threads waiting for requests equal to or slightly greater than the number of processors: it has to keep track of the number of threads waiting for messages, running in the server, and blocked on outgoing RPCs or kernel calls. The last aspect is particularly awkward because it requires surrounding every such blocking call with operations to wake up and manage server threads. If it does not, deadlock can result. Finally, substantial complexity is due to multiplexing `cthreads` over kernel threads, which cannot be done with migrating threads[12].⁸ However, if it is found necessary to limit the total number of executing threads in a particular server, in order to avoid saturation due to excessive kernel context switching, some of this simplification will not be present.

6.3 User-level Thread Issues

The most important issue with migrating RPC is that the user-level threads and synchronization package most widely used on Mach, `cthreads`, has significant limitations in the presence of migrating RPC.

Server Thread Management The `cthreads` library presents a significant problem to the server of a migrating RPC. Servers use `cthreads` to multiplex user threads on top of kernel threads, replacing kernel-mode context switches with much faster user-level context switches, whenever possible. However, one of the main assumptions made by the user-level threads package is that all of the kernel threads on which it is running its user-level threads are interchangeable—that one kernel thread can be used for an operation just as well as another. This assumption can be satisfied in a static thread model, although in the process it makes real-time monitoring and control of server threads difficult.

In a migrating thread model, however, kernel threads migrating in from clients are *not* interchangeable—they may have different priorities and other attributes. Even ignoring this, the return-to-kernel after an RPC has been processed must be done on the same kernel thread that the RPC came in on. In general, trying to multiplex threads in this manner loses one of the main advantages of our design: providing a kernel

⁷When programming to the RPC stub generator interface, as is usually done, this is not a change in binding semantics.

⁸Skeptics should inspect the OSF/1 server's `ux_server_loop` and related code, where the outlined complexity was found necessary for performance and safety.

entity (the activation stack) which represents a particular piece of work in progress, i.e., an entire logical thread of control. Therefore, multiplexing a server's user-level threads on top of incoming kernel threads is not appropriate. In `cthreads`, multiplexing can easily be avoided by "wiring" the user-level thread.

However, some speed is lost in the elimination of user-level thread multiplexing, because synchronization operations in the server sometimes now require kernel-level context switches instead of user-level context switches. Measuring real applications, including on multiprocessors, will be necessary before we can be sure the gains from better RPC performance are not outweighed by this additional cost. We believe that the speed advantage of user-level context switching is not as significant in typical RPC servers as it is in compute-intensive applications, which are the traditional benchmarks for thread implementations. In well-designed servers providing "system" functions, we suspect that internal contention can be minimized so that the importance of RPC speed outweighs that of context switch speed. We point out that in many commercial microkernel-based systems, including QNX[20], Chorus[26], and KeyKOS[6], OS servers do not generally multiplex user-level threads over multiple kernel threads. Instead, these systems either provide multithreading purely with kernel threads, or their functions are sufficiently decomposed so that each server can be based on a single kernel thread, requiring no internal synchronization. However, until there is more extensive performance analysis of servers using migrating RPC, losing user-level threads when servicing RPCs remains a concern.

Note that it is only for "guest" threads migrating in from other tasks that user-level thread multiplexing is a problem; threads native to the server can still use some kind of user-level thread system, or even a specialized multiplexing mechanism such as scheduler activations.

A More Appropriate Synchronization System Since `cthreads` can no longer multiplex user-level threads on kernel threads in servers, it should be replaced with a synchronization library better optimized to provide synchronization over kernel threads. Also, kernel-visible synchronization will be necessary to fully implement priority inheritance, as we discuss in a longer paper[18]. We are planning a replacement for `cthreads` that provides synchronization primitives in a user-level library, but in cooperation with the kernel.

7 The Unix Server

To function on the new kernel using traditional RPC, no changes were necessary to the OSF/1 single server and emulator, or to the libraries they use. To support migrating RPC, a few changes were required. Initially, we chose ways which have minimal impact on existing code, but better, cleaner mechanisms can be provided in the longer term. The server was modified to invoke the new setup function in the `cthreads` library and to wire incoming `cthreads`. The existing complex management of the server's thread pool, while basically no longer used, was retained. We made no modifications to the emulator. Since we are providing backwards-compatible semantics for thread manipulation, no modifications were needed to the existing complex code for handling Unix signals.

Even when our implementation is tuned, we do not anticipate a large performance improvement in the single server, to a large extent because it was written with the assumption that RPC is very expensive. Therefore, the server avoids RPC as much as possible, instead resorting to other approaches like shared memory pages, whose performance is not enhanced by migrating RPC. However, our initial goal is not primarily to show performance improvement, but to demonstrate the gain in simplicity and cleanliness provided by migrating threads, and how migrating threads can be implemented in a backward-compatible way in an existing operating system.

Desirable Modifications We anticipate that the Unix server could be made simpler with two modifications that take advantage of migrating threads. One is emulating Unix signals under Mach, and is described in [18]. Another is that because Mach 3.0 provides no standard way of propagating abort requests into RPCs, the Unix server must manually handle all Unix system call interruptions such as those caused by pending signals. It ought to be considerably simplified by taking advantage of the propagating abort operations now provided by the kernel. This would also make interruption semantics naturally extend to other servers in the system, such as ones installed by Mach-specific application programs running under Unix.

8 Results

8.1 Status

We have completed the kernel implementation of the system described in this paper. An unmodified emulator-based Unix server runs normally on the new microkernel, using traditional thread-switching RPC. A server modified to provide activations, so that it uses migrating RPC, runs multi-user. Unix signals, including ^C and ^Z, are working.

8.2 Experimental Environment

All timings were collected on a single HP9000/730 with 64 MB RAM. This machine has a 67 Mhz PA-RISC 1.1 processor, 128K offchip Icache, 256K offchip Dcache, 96 entry ITLB, and 96 entry DTLB, with a page size of 4K. The caches are direct-mapped and virtually addressed, with a cache miss cost of about 14 cycles. RPC test times were collected by reading the PA's clock register which increments every cycle, and can be read in user mode. Other times were obtained from the Unix server.

The system software is our port of the Mach 3.0 kernel, version NMK14.4, and the emulator-based OSF/1 single server, version 1.0.4b1. The compiler is GCC 2.4.5.u5 with full optimization.

8.3 RPC Path Breakdown and Analysis

To analyze the 3.6 times speedup in null RPC presented in the next section, we counted instructions by hand along the kernel's null RPC path. These counts are broken down by type of processing in Table 1, along with the relative (old/new ratio) and absolute (number of instructions) improvement in each category. 12% of the improvement is due to the inverted server-kernel interface: since the kernel is now "calling" the server rather than vice versa, the kernel no longer needs to save and restore the server's registers on every RPC. 21% results from the kernel's "first-hand" knowledge of RPC: it no longer needs to create, translate, and consume reply ports in order to match a reply to its request. 41% comes directly from the optimized control transfer of migrating threads: switching activations is much simpler than switching threads. 20% of the improvement is due to simpler data management, particularly the elimination of the need to maintain a temporary message buffer in the kernel's address space due to direct copy from source to destination. It is arguable whether this aspect is related to migrating threads.⁹

It is interesting that nearly half of the cost of migrating RPC now resides at the kernel-client boundary (more than half, if measured by memory operations—see below). Therefore, further improvements to other parts of the kernel RPC path will probably lead to only minimal overall speedup. That upcalls are so cheap in comparison, especially in memory operations, may have implications for system structuring in a system supporting migrating threads.

Table 1: Null RPC Path: Breakdown and Improvement

Stage	Instruction Count				Improvement (Instructions)			Improvement (Loads/Stores)	
	Switch		Migrate		$\frac{Sw}{Mg}$	$Sw - Mg$		$Sw - Mg$	
Kernel entry/exit: Client side	146	13%	99	46%	1.5	47	5%	4	1%
Kernel entry/exit: Server side	146	13%	33	15%	4.4	113	12%	58	16%
Port translation	206	18%	12	6%	17.2	194	21%	82	23%
Thread/Activation switch	408	36%	30	14%	13.6	378	41%	152	42%
Message copy	222	20%	39	18%	5.7	183	20%	66	18%
Entire kernel path	1128	100%	213	100%	5.3	915	100%	362	100%

The last two columns of Table 1 show the improvement for each stage, as measured by the number of load/store operations, which should contribute disproportionately to total cycles due to memory subsystem

⁹ At first sight, this aspect may seem completely orthogonal. However, on the switching path, the message copyin is at the very beginning, while the copyout into the destination is at the end, separated by hundreds of lines of complex code. Combining these into one direct copy operation would be very difficult. On the much simpler and shorter migrating path, direct copy was easy to support. Note that even with null RPC some copying is involved: the 6 word message header.

costs. We observe that the percentage improvements in instruction count and memory operations are approximately equal for each stage of RPC. This suggests that instruction count is a valid measure of the relative contribution of each stage to the overall performance gain.

Examination of the context switch code in the old optimized RPC path explains much of its cost: the kernel essentially executes a portion of the scheduler specially hand-coded inline. Numerous constraints must be satisfied: both old and new threads must be in just the right states, run and wait queues must be maintained correctly, locks on ports, threads, IPC spaces, and other data structures must be taken and released in the right order to avoid deadlocks; timers are manipulated; interrupt levels are changed; resources acquired along the way must be carefully tracked to ensure that it will be possible to unroll everything if, for some reason, the computation falls off the optimized path.

Table 2 shows the instruction mix for each path, broken into three categories: total instructions, loads/stores, and branches. The migrating path has a somewhat higher percentage of loads and stores (56% vs. 43%), presumably due to the fact that the basic memory-intensive aspects of IPC—register saving and restoring, memory copying, and data structure traversal—are less obscured by computational overhead. The relative incidence of branch instructions is much lower, however (10% vs. 17%). This, along with the ninefold reduction in total number of branch instructions, reflects the lower logical complexity of the migrating path.

Table 2: Null RPC Path: Instruction Mix

Stage	Switch						Migrate			
	All	Load/Store		Branch		All	Load/Store		Branch	
Kernel entry/exit: Client side	146	66	45%	12	8%	99	62	62%	10	10%
Kernel entry/exit: Server side	146	66	45%	12	8%	33	8	24%	3	9%
Port translation	206	90	44%	45	22%	12	8	67%	1	8%
Thread/Activation switch	408	174	43%	78	19%	30	22	73%	3	10%
Message copy	222	86	39%	46	21%	39	20	51%	5	13%
Entire kernel path	1128	482	43%	193	17%	213	120	56%	22	10%

We expect that our results on the RPC path would, in general, extend to other architectures besides the PA-RISC. Although sometimes difficult, most architectures can achieve the single direct copy when the data are contiguous, for example by temporary mapping[23]. Changing the interrupt priority level (IPL) is done four times on the switching path due to scheduler involvement, but not at all on the migrating path; while IPL changes are cheap on the PA-RISC, they are very expensive on some other architectures[28], making migrating threads especially important on them. The unavoidable cost of address space switching is much higher on some architectures, which would lead to a lower improvement ratio, but even then we expect the benefits to be considerable.

8.4 Micro and Macro Benchmark Results

Measurements of the costs of cross-task migrating and traditional switching RPC are presented in Table 3. The columns on the left include only the kernel costs of RPC, while the ones on the right include both kernel and user (marshaling) costs, obtained in another set of runs. On this machine, a null local RPC now spends less than 10 microseconds in the kernel. The speedup from migrating threads varies with parameter size from a factor of 3.4 for null RPC, a factor of 2.0 for 1K of data, to a factor of 1.7 for long in-line marshaled data. This factor of 1.7 comes from the fact that the data is copied three times in the switching path—once during marshaling and twice in the kernel—but only twice on the complete migrating path.

One interesting observation is that the number of cycles per instruction (CPI) is considerably worse on the migrating path ($3.0 = 648/213$) than on the original path ($2.1 = 2318/1128$). We believe that some or all of this is due to two factors: the higher percentage of load/store instructions as described above, and the fact that the instructions on the hand-coded migrating path were not carefully scheduled and optimized like the C compiler did to most of the old path. Therefore, more careful coding of the migrating RPC path could somewhat lower CPI. More investigation of the CPI difference is warranted.

Table 3: RPC Times in Cycles

Test	Kernel Time			Kernel Time + User Marshaling		
	Switching	Migrating	Ratio	Switching	Migrating	Ratio
Null RPC	2318	648	3.6	2986	880	3.4
32 In	2379	683	3.5	3050	961	3.2
1K In	4753	1676	2.8	6470	3210	2.0
32K In	137808	(34703) (84527)	(4.0) (1.6)	183164	109857	1.7

The measurement of the kernel time for 32K migrating RPC showed severe side effects of the HP730's direct-mapped cache. At the top is our original measurement, a suspiciously low time resulting in a rather unbelievable $4.0\times$ speed improvement. Below it is the same measurement taken after shifting the message buffers slightly so that the cache lines would conflict, resulting in an improvement of $1.6\times$, *below* the factor of two we would expect due to the data being copied once instead of twice. This demonstrates the importance of cache effects in data transfer, and deserves further investigation in the future.

As a preliminary test of overall performance impact, we measured the time for a "make" of the `gas` assembler. Under migrating threads, the elapsed time went from 109 to 107 seconds, an improvement of about 2%. The link phase took about 3 seconds. A link of a larger program (the HP linker itself), improved from 14 seconds to 12 seconds, an improvement of 14%. We believe this greater improvement is due to `ld` having a higher ratio of system calls to computation.

One area where we slow down is in RPCs to the kernel. These do not currently migrate since we haven't changed the kernel to provide activations on its ports. We do not expect doing so to be difficult. When that is done, all messages which originally would have used the optimized path (true RPCs), should be migrating.

We expect a tuned implementation to achieve more overall speedup, and if other RPC optimizations enabled by migrating threads are performed, significantly more speedup.

8.5 Kernel Code Simplification

Confined Controllability Making threads independent of tasks and uncontrollable outside of user mode greatly reduced code complexity in a number of areas. The source file containing most thread control operations was reduced by more than half, from 72K to 32K. In the new 18K source file supporting activations, support for control operations account for only about 10K.¹⁰ This simplification largely resulted from cleaner management of thread suspension, resumption, and termination. The original Mach 3.0 thread control mechanisms had to make numerous tests for special cases, such as a thread manipulating itself, or two threads trying to control each other simultaneously, possibly causing kernel deadlock. Now that controllability is confined within well-defined boundaries, as it must be to support migrating threads, these tricky cases never occur because kernel code is always "out of bounds."

The task management code was reduced from 38K to 20K for similar reasons: the cleaner model simplified locking and eliminated many special-case situations such as the case of a thread terminating its own task.

Migrating RPC On the original switching path, the port translation and context switch code were mostly written in machine-independent C code, while the other categories were PA-specific assembly language. On the migrating RPC path, the machine-independent parts became so trivial that it was easier to inline them into the assembly language path than to go to the trouble of interfacing with a high-level language.¹¹ The entire 1350 lines of complex C code comprising the optimized RPC path plus about 400 hand-coded assembler instructions, were replaced with about 220 assembler instructions. The resultant simplifications in logical complexity, a factor of nine, are evident from the figures presented in Section 8.3.

¹⁰The rest is for allocation and freeing of activations and other administration, unrelated to the control operations.

¹¹Although some speedup presumably resulted from assembly coding, we believe it was slight, and only feasible due to the simplicity of the migrating path. Those who differ are invited to inspect the original message path and try to hand-code it without changing its semantics.

8.6 Memory Use

In the original microkernel, in general only a few kernel stacks (8K each) were required per processor, due to the continuations mechanism[14]. At the beginning of this project, we disabled continuations in order to simplify our work; this immediately raised kernel memory use to one kernel stack per thread. However, with migrating threads there are now far fewer threads in the system, and kernel stacks are still associated with threads instead of activations. While running our multiuser benchmarks, we observe kernel physical memory use to be at most 300K greater than in the original system. However, we are in the process of reintroducing continuations under the new model, so even this increase should be temporary.

Regarding server virtual memory use, at this writing we statically allocate a large number of activations (40) and the old thread pool remains in the server. Therefore, about three times as much VM is used as before (2.6 MB vs. .8 MB) When we remove the thread pool, server VM use should be about the same as in the old system, because for the most part each server thread/user stack becomes one server activation/user stack. Of course, clients are unaffected because they are unmodified.

9 Future Work

This work *enables* many further improvements to Mach and Mach servers, as well as raising areas for further research. Providing an appropriate replacement for the `cthreads` synchronization primitives is important in order to make a fair evaluation of the impact of relying on kernel-level context switches. Our earlier work on moving trusted servers into the kernel's protection domain and address space (INKS)[22] used ad-hoc thread migration. By re-working the thread abstraction from scratch, our new system solves all of the problems encountered[17].

The "NORMA" (NO Remote Memory Access)[2] version of Mach 3.0 allows IPC between different nodes of a distributed memory multiprocessor, implemented in the microkernel. Extending the migrating thread system to encompass RPC between nodes should be done. The issues involved have already been explored in depth in Alpha[11].

Going in a different direction, our work allows improvements in Mach's support for real-time systems. At the implementation level, we have largely decoupled two portions of the thread abstraction: the schedulable entity (priority, scheduling policies, etc.) from the thread of control (the chain of activations). This makes it feasible to decouple them entirely, enabling a full implementation of priority inheritance.

The Mach message format imposes unnecessary overhead on migrating RPC. The migrating thread model enables other designs which could provide much higher performance, such as LRPC[3]. In cases where protection domains have been merged[22], much of the copying can be avoided.

The migrating RPC mechanism can also be used in thread exception processing. This will allow a no-emulator server, such as OSF/1-MK5[25], to do more efficient argument copying. We believe migrating RPC can also be leveraged by making the Mach pager interface synchronous, with a thread servicing its own page faults. This requires security to be explicitly provided when untrusted pagers are involved.

The OSF Research Institute is adopting our code and is planning to make many of the above improvements, in an Intel 486 base. Our code will also be available to interested parties.

10 Conclusion

We draw three main conclusions from our work. First, by changing the thread model of an existing operating system, and evaluating the two versions, we show that a migrating thread model is superior to a static model. Migrating threads provide superior functionality, performance, and code simplification. In the area of functionality, thread migration (i) provides more precisely defined semantics for thread manipulation and additional control operations, (ii) allows scheduling and other attributes to follow threads, especially important for real-time systems. In performance, thread migration (i) improves the performance of ordinary RPC, and (ii) enables a multitude of aggressive RPC optimizations, especially in systems under current research which provide cross-domain memory or address-space sharing. However, thread migration does have the potential performance disadvantage of not allowing user-level threads that service RPCs to be multiplexed atop multiple kernel threads. In reducing implementation complexity, thread migration simplifies (i) kernel

code, and, we expect, (ii) server code. In each of these areas, our implementation and measurements have demonstrated the first benefit, while potential gains from the second seem evident, but have not yet been realized through full implementation in our system.

Secondly, since migrating threads requires that the kernel treat local RPC as an identifiable semantic entity, we conclude that operating system kernels should directly support the RPC abstraction.

Our third main conclusion is that it is feasible to improve at least some *existing* operating systems, by changing their thread model from static to migrating. Even in the case of Mach 3.0, which has an unusually rich thread-manipulation interface, we show that this far-reaching change can be made while retaining backward compatibility, and with only moderate implementation effort. A key element of that implementation is “basing” threads in the kernel, which temporarily make excursions into tasks via upcalls.

Acknowledgements

We especially thank Mike Hibler for his expert help with the implementation, as well as discussion of controllability and signal issues. We thank Douglas Orr for varied help and Greg Minshall for his careful review of an earlier draft. We had helpful discussions with many members of the OSF Research Institute about many aspects of the system, and with the HP Labs Brevix group about controllability. The anonymous referees, Brian Bershad, Rich Draves, and Alessandro Forin provided many useful comments on earlier drafts, and we thank Mike Jones and Jeff Mogul for all of that, plus their patient shepherding.

References

- [1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.
- [2] J. S. Barrera. A fast Mach network IPC implementation. In *Proc. of the Second USENIX Mach Symposium*, pages 1–12, 1991.
- [3] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.
- [4] Andrew D. Birrell. An introduction to programming with threads. Technical Report SRC-35, DEC Systems Research Center, January 1989.
- [5] A. P. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Trans on Software Engineering*, SE-13(1):65–76, 1987.
- [6] Alan C. Bomberger and Norman Hardy. The KeyKOS nanokernel architecture. In *Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 95–112, Seattle, WA, April 1992.
- [7] John B. Carter, Bryan Ford, Mike Hibler, Ravindra Kuramkote, Jeffrey Law, Jay Lepreau, Douglas B. Orr, Leigh Stoller, and Mark Swanson. FLEX: A tool for building efficient and flexible systems. In *Proc. Fourth Workshop on Workstation Operating Systems*, October 1993.
- [8] D. R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.
- [9] Roger S. Chin and Samuel T. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1), March 1991.
- [10] David D. Clark. The structuring of systems using upcalls. In *Proc. of the 10th ACM Symposium on Operating Systems Principles*, pages 171–180, Orcas Island, WA, December 1985.
- [11] Raymond K. Clark, E. Douglas Jensen, and Franklin D. Reynolds. An architectural overview of the Alpha real-time distributed kernel. In *Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 127–146, Seattle, WA, April 1992.
- [12] Michael Conduct. Personal communication, November 1993.
- [13] Sadegh Davari and Lui Sha. Sources of unbounded priority inversions in real-time systems and a comparative study of possible solutions. *ACM Operating Systems Review*, 23(2):110–120, April 1992.

- [14] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, Asilomar, CA, October 1991.
- [15] Peter Druschel, Larry L. Peterson, and Norman C. Hutchinson. Beyond micro-kernel design: Decoupling modularity and protection in Lipto. In *Proc. of the 12th International Conference on Distributed Computing Systems*, pages 512–520, Yokohama, Japan, June 1992.
- [16] Partha Dasgupta et al. The design and implementation of the Clouds distributed operating system. *Computing Systems*, 3(1), Winter 1990.
- [17] Bryan Ford, Mike Hibler, and Jay Lepreau. Notes on thread models in Mach 3.0. Technical Report UUCS-93-012, University of Utah Computer Science Department, April 1993.
- [18] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to use migrating threads. Technical Report UUCS-93-022, University of Utah, November 1993.
- [19] Graham Hamilton and Panos Kougouris. The Spring nucleus: a microkernel for objects. In *Proc. of the Summer 1993 USENIX Conference*, pages 147–159, Cincinnati, OH, June 1993.
- [20] Dan Hildebrand. An architectural overview of QNX. In *Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, April 1992.
- [21] D.B. Johnson and W. Zwaenepoel. The Peregrine high-performance RPC system. *Software — Practice and Experience*, 23(2):201–221, February 1993.
- [22] Jay Lepreau, Mike Hibler, Bryan Ford, and Jeff Law. In-kernel servers on Mach 3.0: Implementation and performance. In *Proc. of the Third USENIX Mach Symposium*, pages 39–55, April 1993.
- [23] Jochen Liedtke. Improving IPC by kernel design. In *Proc. of the 14th ACM Symposium on Operating Systems Principles*, Asheville, NC, December 1993.
- [24] Open Systems Foundation and Carnegie Mellon Univ. *MACH 3 Kernel Interface*, 1992.
- [25] Simon Patience. Redirecting system calls in Mach 3.0: An alternative to the emulator. In *Proc. of the Third USENIX Mach Symposium*, pages 57–73, Santa Fe, NM, April 1993.
- [26] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. The Chorus distributed operating system. *Computing Systems*, 1(4):287–338, December 1989.
- [27] Michael L. Scott, Thomas J. LeBlanc, and Brian D. Marsh. Design rationale for Psyche, a general-purpose multiprocessor operating system. In *Proc. of the 1988 International Conference on Parallel Processing*, pages 255–262, August 1988.
- [28] Daniel Stodolsky, J. Bradley Chen, and Brian N. Bershad. Fast interrupt priority management in operating system kernels. In *Proc. of the Second USENIX Workshop on Micro-kernels and Other Kernel Architectures*, San Diego, CA, September 1993.
- [29] Vrije Universiteit, Amsterdam, NL. *The Amoeba 5.0 Reference Manual: Programming Guide*, 1992. *rpc* manual page; ftp.cs.vu.nl:amoeba/manuals/pro.ps.Z.

Author Information

Bryan Ford is an undergraduate in Computer Science at the University of Utah. His current major research interest is improving Mach 3.0, but he pursues other interests, including data compression, languages, graphics, and music. He is the author of several widely used packages for the Amiga, including the XPK compression package and the MultiPlayer music program. Bryan is the designer and primary implementor of Mach migrating threads.

Jay Lepreau is Assistant Director of the Center for Software Science, a research group within Utah's Computer Science Department which works in many aspects of systems software. He has worked with Unix since 1979, and has served as co-chair of the 1984 USENIX conference and on numerous other USENIX program committees. His group has made significant contributions to the BSD and GNU software distributions. His current research interests include flexible system structuring, with operating system, language, linking, and runtime components.

The author's addresses are: Center for Software Science, Department of Computer Science, University of Utah, 84112. They can be reached electronically at {baford,lepreau}@cs.utah.edu.

Unix is a trademark of USL. OSF/1 is a trademark of the Open Software Foundation. OS/2 is a trademark of IBM.

TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems

Pete Keleher, Alan L. Cox, Sandhya Dwarkadas and Willy Zwaenepoel

Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract

TreadMarks is a *distributed shared memory* (DSM) system for standard Unix systems such as SunOS and Ultrix. This paper presents a performance evaluation of TreadMarks running on Ultrix using DECstation-5000/240's that are connected by a 100-Mbps switch-based ATM LAN and a 10-Mbps Ethernet. Our objective is to determine the efficiency of a user-level DSM implementation on commercially available workstations and operating systems.

We achieved good speedups on the 8-processor ATM network for Jacobi (7.4), TSP (7.2), Quicksort (6.3), and ILINK (5.7). For a slightly modified version of Water from the SPLASH benchmark suite, we achieved only moderate speedups (4.0) due to the high communication and synchronization rate. Speedups decline on the 10-Mbps Ethernet (5.5 for Jacobi, 6.5 for TSP, 4.2 for Quicksort, 5.1 for ILINK, and 2.1 for Water), reflecting the bandwidth limitations of the Ethernet. These results support the contention that, with suitable networking technology, DSM is a viable technique for parallel computation on clusters of workstations.

To achieve these speedups, TreadMarks goes to great lengths to reduce the amount of communication performed to maintain memory consistency. It uses a lazy implementation of release consistency, and it allows multiple concurrent writers to modify a page, reducing the impact of false sharing. Great care was taken to minimize communication overhead. In particular, on the ATM network, we used a standard low-level protocol, AAL3/4, bypassing the TCP/IP protocol stack. Unix communication overhead, however, remains the main obstacle in the way of better performance for programs like Water. Compared to the Unix communication overhead, memory management cost (both kernel and user level) is small and wire time is negligible.

1 Introduction

With increasing frequency, networks of workstations are being used as parallel computers. High-speed general-purpose networks and very powerful workstation processors have narrowed the performance gap between workstation clusters and supercomputers. Furthermore, the workstation approach provides a relatively low-cost, low-risk entry into the parallel computing arena. Many organizations already have an installed workstation base, no special hardware is required to use this facility as a parallel computer, and the resulting system can be easily maintained, extended and upgraded. We expect that the workstation cluster approach to parallel computing will gain further popularity, as advances in networking continue to improve its cost/performance ratio.

This research was supported in part by the National Science Foundation under Grants CCR-9116343, CCR-9211004, CDA-9222911, and CDA-9310073, by the Texas Advanced Technology Program under Grant 003604014, and by a NASA Graduate Fellowship.

Various software systems have been proposed and built to support parallel computation on workstation networks, e.g., tuple spaces [2], distributed shared memory [18], and message passing [23]. TreadMarks is a *distributed shared memory* (DSM) system [18]. DSM enables processes on different machines to share memory, even though the machines physically do not share memory (see Figure 1). This approach is attractive since most programmers find it easier to use than a message passing paradigm, which requires them to explicitly partition data and manage communication. With a global address space, the programmer can focus on algorithmic development rather than on managing partitioned data sets and communicating values.

Many DSM implementations have been reported in the literature (see [20] for an overview). Unfortunately, none of these implementations are widely available. Many run on in-house research platforms, rather than on generally available operating systems, or require kernel modifications that make them unappealing. Early DSM systems also suffered from performance problems. These early designs implemented the shared memory abstraction by imitating consistency protocols used by hardware shared memory multiprocessors. Given the large consistency units in DSM (virtual memory pages), false sharing was a serious problem for many applications.

TreadMarks overcomes most of these problems: it is an efficient DSM system that runs on commonly available Unix systems. This paper reports on an implementation on Ultrix using 8 DECStation-5000/240s, connected both by a 100-Mbps point-to-point ATM LAN and by a 10-Mbps Ethernet. The system has also been implemented on SunOS using SPARCstation-1's and -2's connected by a 10-Mbps Ethernet. The implementation is done at the *user* level, without modification to the operating system kernel. Furthermore, we do not rely on any particular compiler. Instead, our implementation relies on (user-level) memory management techniques to detect accesses and updates to shared data. In order to address the performance problems with earlier DSM systems, the TreadMarks implementation focuses on reducing the amount of communication necessary to keep the distributed memories consistent. It uses a lazy implementation [14] of release consistency [13] and multiple-writer protocols to reduce the impact of false sharing [8].

On the 100-Mbps ATM LAN, good speedups were achieved for Jacobi, TSP, Quicksort, and ILINK (a program from the genetic LINKAGE package [16]). TreadMarks achieved only a moderate speedup for a slightly modified version of the Water program from the SPLASH benchmark suite [22], because of the high synchronization and communication rates. We present a detailed decomposition of the overheads. For the applications measured, the software communication overhead is the bottleneck in achieving high performance for finer grained applications like Water. This is the case even when using a low-level adaptation layer protocol (AAL3/4) on the ATM network, bypassing the TCP/IP protocol stack. The communication overhead dominates the memory management and consistency overhead. On a 100-Mbps ATM LAN, the "wire" time is all but negligible.

The outline of the rest of this paper is as follows. Section 2 focuses on the principal design decisions: release consistency, lazy release consistency, multiple-writer protocols, and lazy diff creation. Section 3

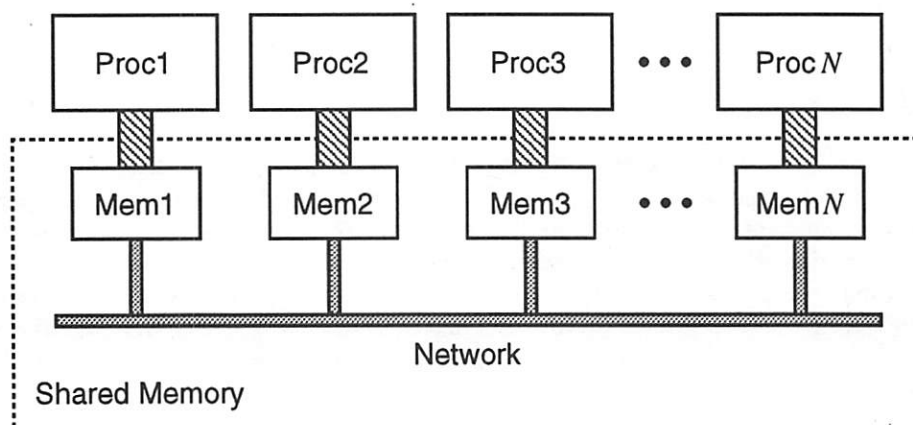


Figure 1 Distributed Shared Memory

describes the implementation of these concepts, and also includes a discussion of the Unix aspects of the implementation. The resulting performance is discussed in Section 4, and compared against earlier work using eager release consistency in Section 5. We discuss related work in Section 6, and conclude in Section 7.

2 Design

TreadMarks' design focuses on reducing the amount of communication necessary to maintain memory consistency. To this end, it presents a *release consistent* memory model [13] to the user. Release consistency requires less communication than conventional, sequentially consistent [15] shared memory, but provides a very similar programming interface. The *lazy* implementation of release consistency in TreadMarks further reduces the number of messages and the amount of data compared to earlier, eager implementations [8]. False sharing is another source of frequent communication in DSM systems. TreadMarks uses *multiple-writer* protocols to address this problem. Multiple-writer protocols require the creation of *diffs*, data structures that record updates to parts of a page. With lazy release consistency, diff creation can often be postponed or avoided, a technique we refer to as *lazy diff creation*.

2.1 Release Consistency

Release consistency (RC) [13] is a *relaxed* memory consistency model that permits a processor to delay making its changes to shared data visible to other processors until certain synchronization accesses occur. Shared memory accesses are categorized either as *ordinary* or as *synchronization* accesses, with the latter category further divided into *acquire* and *release* accesses. Acquires and releases roughly correspond to synchronization operations on a lock, but other synchronization mechanisms can be implemented on top of this model as well. For instance, arrival at a barrier can be modeled as a release, and departure from a barrier as an acquire. Essentially, RC requires ordinary shared memory updates by a processor p to become visible at another processor q , only when a subsequent release by p becomes visible at q .

In contrast, in *sequentially consistent* (SC) memory [15], the conventional model implemented by most snoopy-cache, bus-based multiprocessors, modifications to shared memory must become visible to other processors immediately [15]. Programs written for SC memory produce the same results on an RC memory, provided that (i) all synchronization operations use system-supplied primitives, and (ii) there is a release-acquire pair between conflicting ordinary accesses to the same memory location on different processors [13]. In practice, most shared memory programs require little or no modifications to meet these requirements.

Although execution on an RC memory produces the same results as on a SC memory for the overwhelming majority of the programs, RC can be implemented more efficiently than SC. In the latter, the requirement that shared memory updates become visible immediately implies communication on each write to a shared data item for which other cached copies exist. No such requirement exists under RC. The propagation of the modifications can be postponed until the next synchronization operation takes effect.

2.2 Lazy Release Consistency

In *lazy release consistency* (LRC) [14], the propagation of modifications is postponed *until the time of the acquire*. At this time, the acquiring processor determines which modifications it needs to see according to the definition of RC.

To do so, LRC divides the execution of each process into *intervals*, each denoted by an *interval index*. Every time a process executes a release or an acquire, a new interval begins and the interval index is incremented. Intervals of different processes are partially ordered [1]: (i) intervals on a single processor are totally ordered by program order, and (ii) an interval on processor p precedes an interval on processor q if the interval of q begins with the acquire corresponding to the release that concluded the interval of p . This partial order can be represented concisely by assigning a *vector timestamp* to each interval. A vector timestamp contains an entry for each processor. The entry for processor p in the vector timestamp of interval i of processor p is equal to i . The entry for processor $q \neq p$ denotes the most recent interval of processor q that precedes the current interval of processor p according to the partial order. A processor computes a new vector timestamp at an acquire according to the pair-wise maximum of its previous vector timestamp and the releaser's vector timestamp.

RC requires that before a processor p may continue past an acquire, the updates of all intervals with a smaller vector timestamp than p 's current vector timestamp must be visible at p . Therefore, at an acquire, p sends its current vector timestamp to the previous releaser, q . Processor q then piggybacks on the release-acquire message to p , *write notices* for all intervals named in q 's current vector timestamp but not in the vector timestamp it received from p .

A write notice is an indication that a page has been modified in a particular interval, but it does *not* contain the actual modifications. The timing of the actual data movement depends on whether an invalidate, an update, or a hybrid protocol is used (see [9]). TreadMarks currently uses an invalidate protocol: the arrival of a write notice for a page causes the processor to invalidate its copy of that page. A subsequent access to that page causes an access miss, at which time the modifications are propagated to the local copy.

Alternative implementations of RC generally cause more communication than LRC. For example, the DASH shared-memory multiprocessor [17] implements RC in hardware, buffering writes to avoid blocking the processor until the write has been performed with respect to main memory and remote caches. A subsequent release is not allowed to perform (i.e., the corresponding lock cannot be granted to another processor) until all outstanding shared writes are acknowledged. While this strategy masks latency, LRC sends far fewer messages, an important consideration in a software implementation on a general-purpose network because of the high per message cost. In an *eager* software implementation of RC [8], a processor propagates its modifications of shared data when it executes a release. This approach also leads to more communication, because it requires a message to be sent to all processors that cache the modified data, while LRC propagates the data only to the next acquirer.

2.3 Multiple-Writer Protocols

False sharing was a serious problem for early DSM systems. It occurs when two or more processors access different variables within a page, with at least one of the accesses being a write. Under the common single-writer protocols, false sharing leads to unnecessary communication. A write to any variable of a page causes the entire page to become invalid on all other processors that cache the page. A subsequent access on any of these processors incurs an access miss and causes the modified copy to be brought in over the network, although the original copy of the page would have sufficed, since the write was to a variable different from the one that was accessed locally. This problem occurs in snoopy-cache multiprocessors as well, but it is more prevalent in software DSM because the consistency protocol operates on pages rather than smaller cache blocks.

To address this problem, Munin introduced a *multiple-writer* protocol [8]. With multiple-writer protocols two or more processors can simultaneously modify their local copy of a shared page. Their modifications are merged at the next synchronization operation in accordance with the definition of RC, thereby reducing the effect of false sharing.

2.4 Lazy Diff Creation

In order to capture the modifications to a shared page, it is initially write-protected. At the first write, a protection violation occurs. The DSM software makes a copy of the page (a *twin*), and removes the write protection so that further writes to the page can occur without any DSM intervention. The twin and the current copy can later be compared to create a *diff*, a runlength encoded record of the modifications to the page.

In TreadMarks, diffs are only created when a processor requests the modifications to a page or a write notice from another processor arrives for that page. In the latter case, it is essential to make a diff in order to distinguish the modifications made by the different processors. This *lazy* diff creation is distinct from Munin's implementation of multiple-writer protocols, where at each release a diff is created for each modified page and propagated to all other copies of the page. The lazy implementation of RC used by TreadMarks allows diff creation to be postponed until the modifications are requested. Lazy diff creation results in a decrease in the number of diffs created (see Section 5) and an attendant improvement in performance.

3 Implementation

3.1 Data Structures

Figure 2 gives an overview of the data structures used. The principal data structures are the *PageArray*, with one entry for each shared page, the *ProcArray*, with one entry for each processor, a set of *interval records* (containing mainly the vector timestamp for that interval), a set of *write notice records*, and a *diff pool*. Each entry in the *PageArray* contains:

1. The current state: no access, read-only access, or read-write access.
2. An *approximate copyset* specifying the set of processors that are believed to currently cache this page.
3. For each page, an array indexed by processor of head and tail pointers to a linked list of *write notice records* corresponding to write notices received from that processor for this page. If the diff corresponding to the write notice has been received, then a pointer to this diff is present in the write notice record. This list is maintained in order of decreasing interval indices.

Each entry in *ProcArray* contains a pointer to the head and the tail of a doubly linked list of *interval records*, representing the intervals of that processor that the local processor knows about. This list is also maintained in order of decreasing interval indices. Each of these interval records contains a pointer to a list of write notice records for that interval, and each write notice record contains a pointer to its interval record.

3.2 Interval and Diff Creation

Logically, a new interval begins at each release and acquire. In practice, interval creation can be postponed until we communicate with another process, avoiding overhead if a lock is reacquired by the same processor. When a lock is released to another processor, or at arrival at a barrier, a new interval is created containing

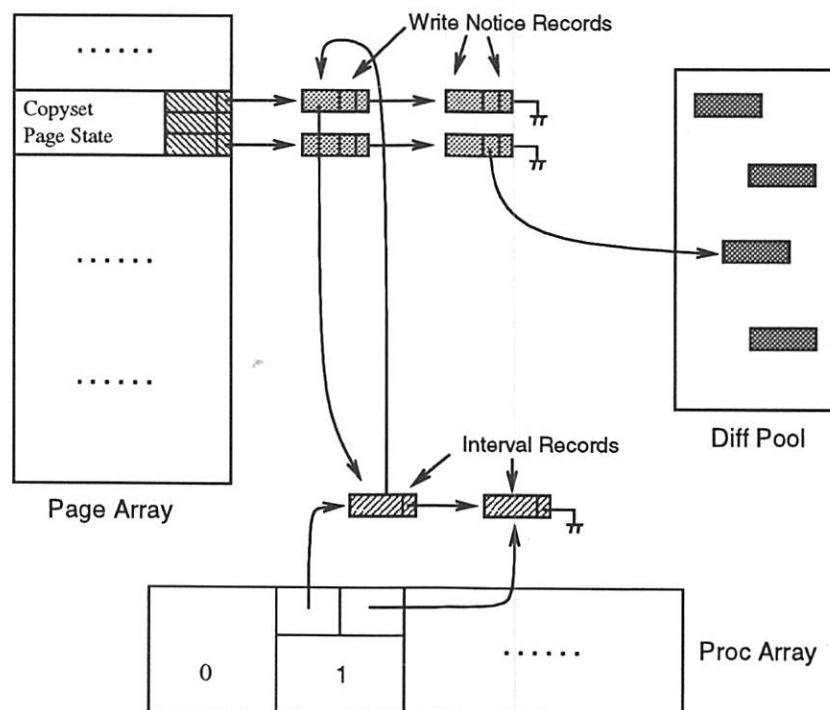


Figure 2 Overview of TreadMarks Data Structures

a write notice for each page that was twinned since the last remote synchronization operation. With lazy diff creation these pages remain writable until a diff request or a write notice arrives for that page. At that time, the actual diff is created, the page is read protected, and the twin is discarded. A subsequent write results in a write notice for the next interval.

3.3 Locks

All locks have a statically assigned manager. Lock management is assigned in a round-robin fashion among the processors. The manager records which processor has most recently requested the lock. All lock acquire requests are directed to the manager, and, if necessary, forwarded to the processor that last requested the lock.

The lock acquire request contains the current vector timestamp of the acquiring processor. The lock request arrives at the processor that either holds the lock or did the last release on it, possibly after forwarding by the lock manager. When the lock is released, the releaser “informs” the acquirer of all intervals between the vector timestamp in the acquirer’s lock request message, and the releaser’s current vector timestamp. The message contains the following information for each of these intervals:

1. The processor id.
2. The vector timestamp.
3. All write-notices. The write notice in the message is a fixed 16-bit entry containing the page number.

All of this information can easily be derived by following the pointers from the *ProcArray* to the appropriate interval records and from there to the appropriate write notice records.

After receiving this message, the acquirer “incorporates” this information into its data structures. For each interval in the message,

1. the acquirer appends an interval record to the interval record list for that processor, and
2. for each write notice
 - (a) it prepends a write notice record to the page’s write notice record list, and
 - (b) adds pointers from the write notice record to the interval record, and vice versa.

Incorporating this information invalidates the pages for which write notices were received.

3.4 Barriers

Barriers have a centralized manager. At barrier arrival, each client “informs” the barrier manager of its vector timestamp and all of the client’s intervals between the last vector timestamp of the manager that the client is aware of (found at the head of the interval record list for the *ProcArray* entry for the manager) and the client’s current vector timestamp. When the manager arrives at the barrier, it “incorporates” these intervals into its data structures. When all barrier arrival messages have been received, the manager then “informs” all clients of all intervals between their vector timestamp, as received in their barrier arrival message, and the manager’s current vector timestamp. The clients then “incorporate” this information as before. As for locks, incorporating this information invalidates the pages for which write notices were received.

3.5 Access Misses

If the faulting processor does not have a copy of the page, it requests a copy from a member of the page’s approximate copyset. The approximate copyset for each page is initialized to contain processor 0.

If write notices are present for the page, the faulting processor obtains the missing diffs and applies them to the page. The missing diffs can be found easily following the linked list of write notices starting from the entry for this page in the *PageArray*. The following optimization minimizes the number of messages necessary to get the diffs. If processor *p* has modified a page during interval *i*, then *p* must have all the diffs of all intervals (including those from processors other than *p*) that have a smaller vector timestamp than *i*. It

therefore suffices to look at the largest interval of each processor for which we have a write notice but no diff. Of that subset of the processors, a message needs to be sent only to those processors for which the vector timestamp of their most recent interval is not dominated by the vector timestamp of another processor's most recent interval.

After the set of necessary diffs and the set of processors to query have been determined, the faulting processor sends out requests for the diffs in parallel, including the processor id, the page number and the interval index of the requested diffs. When all necessary diffs have been received, they are applied in increasing vector timestamp order.

3.6 Garbage Collection

Garbage collection is necessary to reclaim the space used by write notice records, interval records, and diffs. During garbage collection, each processor validates its copy of every page that it has modified. All other pages, all interval records, all write notice records and all diffs are discarded. In addition, each processor updates the copyset for every page. If, after garbage collection, a processor accesses a page for which it does not have a copy, it requests a copy from a processor in the copyset.

The processors execute a barrier-like protocol, in which processors request and apply all diffs created by other processors for the pages they have modified themselves. Garbage collection is triggered when the amount of free space for consistency information drops below a threshold. An attempt is made to make garbage collection coincide with a barrier, since many of the operations are similar.

3.7 Unix Aspects

TreadMarks relies on Unix and its standard libraries to accomplish remote process creation, interprocessor communication, and memory management. In this section, we briefly describe the implementation of each of these services.

TreadMarks interprocessor communication can be accomplished either through UDP/IP on an Ethernet or an ATM LAN, or through the AAL3/4 protocol on the ATM LAN. AAL3/4 is a connection-oriented, unreliable message protocol specified by the ATM standard. Since neither protocol guarantees reliable delivery, TreadMarks uses operation-specific, user-level protocols on top of UDP/IP and AAL3/4 to insure delivery.

To minimize latency in handling incoming asynchronous requests, TreadMarks uses a SIGIO signal handler. Message arrival at any socket used to receive request messages generates a SIGIO signal. Since AAL3/4 is a connection-oriented protocol, there is a socket corresponding to each of the other processors. To determine which socket holds the incoming request, the handler for AAL3/4 performs a `select` system call. The handler for UDP/IP avoids the `select` system call by multiplexing all of the other processors over a single receive socket. After the handler receives the message, it performs the request and returns.

To implement the consistency protocol, TreadMarks uses the `mprotect` system call to control access to shared pages. Any attempt to perform a restricted access on a shared page generates a SIGSEGV signal. The SIGSEGV signal handler examines the local *PageArray* to determine the page's state. If the local copy is read-only, the handler allocates a page from the pool of free pages and performs a `bcopy` to create a *twin*. Finally, the handler upgrades the access rights to the original page and returns. If the local page is invalid, the handler executes the access miss procedure.

4 Performance

4.1 Experimental Environment

Our experimental environment consists of 8 DECstation-5000/240's running Ultrix V4.3. Each machine has a Fore ATM interface that is connected to a Fore ATM switch. The connection between the interface boards and the switch operates at 100-Mbps; the switch has an aggregate throughput of 1.2-Gbps. The interface board does programmed I/O into transmit and receive FIFOs, and requires fragmentation and reassembly of ATM cells by software. Interrupts are raised at the end of a message or a (nearly) full receive FIFO. All of

the machines are also connected by a 10-Mbps Ethernet. Unless otherwise noted, the performance numbers describe 8-processor executions on the ATM LAN using the low-level adaptation layer protocol AAL3/4.

4.2 Basic Operation Costs

The minimum roundtrip time using send and receive for the smallest possible message is 500 μ seconds. The minimum time to send the smallest possible message through a socket is 80 μ seconds, and the minimum time to receive this message is 80 μ seconds. The remaining 180 μ seconds are divided between wire time, interrupt processing and resuming the processor that blocked in receive. Using a signal handler to receive the message at both processors, the roundtrip time increases to 670 μ seconds.

The minimum time to remotely acquire a free lock is 827 μ seconds if the manager was the last processor to hold the lock, and 1149 μ seconds otherwise. In both cases, the reply message from the last processor to hold the lock does not contain any write notices (or diffs). The time to acquire a lock increases in proportion to the number of write notices that must be included in the reply message. The minimum time to perform an 8 processor barrier is 2186 μ seconds. A remote page fault, to obtain a 4096 byte page from another processor takes 2792 μ seconds.

4.3 Applications

We used five programs in this study: Water, Jacobi, TSP, Quicksort, and ILINK. Water, obtained from SPLASH [22], is a molecular dynamics simulation. We made one simple modification to the original program to reduce the number of lock accesses. We simulated 343 molecules for 5 steps. Jacobi implements a form of Successive Over-Relaxation (SOR) with a grid of 2000 by 1000 elements. TSP uses a branch-and-bound algorithm to solve the traveling salesman problem for a 19-city tour. Quicksort sorts an array of 256K integers, using a bubblesort to sort subarrays of less than 1K elements. ILINK, from the LINKAGE package [16], performs genetic linkage analysis (see [10] for more details). ILINK's input consists of data on 12 families with autosomal dominant nonsyndromic cleft lip and palate (CLP).

4.4 Results

Figure 3 presents speedups for the five applications. The speedups were calculated using uniprocessor times obtained by running the applications without TreadMarks. Figure 4 provides execution statistics for each of the five applications when using 8 processors.

The speedup for Water is limited by the high communication (798 Kbytes/second and 2238 messages/second) and synchronization rate (582 lock accesses/second). There are many short messages (the average message size is 356 bytes), resulting in a large communication overhead. Each molecule is protected by a lock that is accessed frequently by a majority of the processors. In addition, the program uses barriers for synchronization.

Jacobi exclusively uses barriers for synchronization. Jacobi's computation to communication ratio is an order of magnitude larger than that of Water. In addition, most communication occurs at the barriers and between neighbors. On the ATM network, this communication can occur in parallel. The above two effects compound, resulting in near-linear speedup for Jacobi.

TSP is an application that exclusively uses locks for synchronization. Like Jacobi, TSP has a very high computation to communication ratio, resulting in near-linear speedup. While the number of messages per second is slightly larger than for Jacobi, TSP transmits only a quarter of the amount of data transmitted by Jacobi.

Quicksort also uses locks for synchronization. Quicksort's synchronization rate is close to that of Jacobi's. It, however, sends over twice as many messages and data per second, resulting in slightly lower, although good, speedups. The number of kilobytes per second transmitted by Quicksort is similar to that transmitted by Water, but it sends 3 times fewer messages and the number of synchronization operations is an order of magnitude lower than for Water. As a result, speedup for Quicksort is higher than for Water.

ILINK achieves less than linear speedup on TreadMarks because of a load balancing problem inherent to the nature of the algorithm [10]. It is not possible to predict in advance whether the set of iterations distributed to the processors will result in the same amount of work on each processor, without significant

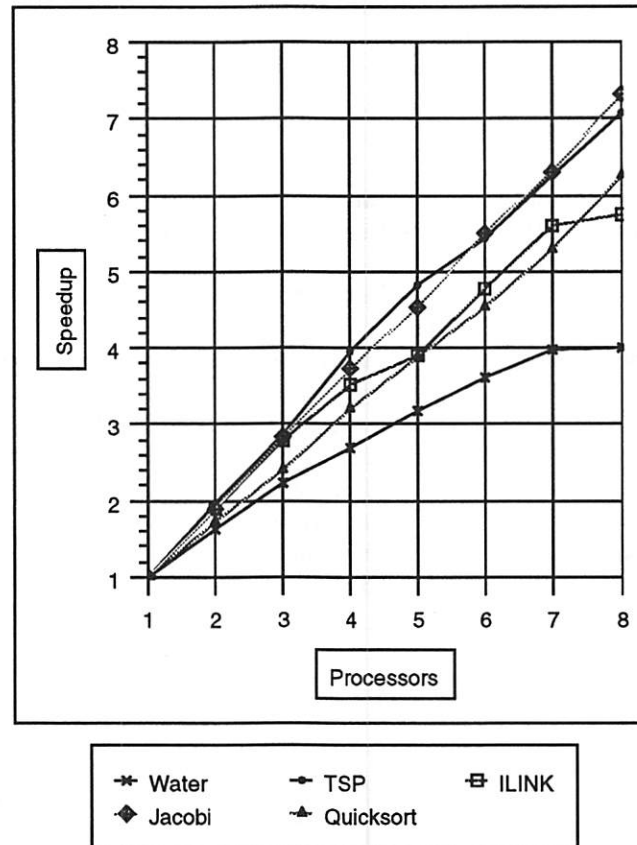


Figure 3 Speedups Obtained on TreadMarks

	Water	Jacobi	TSP	Quicksort	ILINK
Input	343 mols 5 steps	2000x1000 floats	19-city tour	256000 integers	CLP
Time (secs)	15.0	32.0	43.8	13.1	1113
Barriers/sec	2.5	6.3	0	0.4	0.4
Locks/sec	582.4	0	16.1	53.9	0
Msgs/sec	2238	334	404	703	456
Kbytes/sec	798	415	121	788	164

Figure 4 Execution Statistics for an 8-Processor Run on TreadMarks

computation and communication. Consequently, speedups are somewhat lower than one would expect based on the communication and synchronization rates.

4.5 Execution Time Breakdown

Figure 5 shows a percentage breakdown of the execution times for 8-processor versions of all 5 applications. The "Computation" category is the time spent executing application code; "Unix" is the time spent executing Unix kernel and library code; and "TreadMarks" is the time spent executing TreadMarks code. "Idle Time" refers to the time that the processor is idle. Idle time results from waiting for locks and barriers, as well as from remote communication latency.

The largest overhead components are the Unix and idle times. The idle time reflects to some extent the amount of time spent waiting for Unix and TreadMarks operations on other nodes. The TreadMarks overhead is much smaller than the Unix overhead. The largest percentage TreadMarks overhead is for Water (2.9% of overall execution time). The Unix overhead is at least three times as large as the TreadMarks overhead for all the applications, and is 9 times larger for ILINK.

Figure 6 shows a breakdown of the Unix overhead. We divide Unix overhead into two categories: communication and memory management. Communication overhead is the time spent executing *kernel* operations to support communication. Memory management overhead is the time spent executing *kernel* operations to support the *user-level* memory management, primarily page protection changes. In all cases, at least 80% of the kernel execution time is spent in the communication routines, suggesting that cheap communication is the primary service a software DSM needs from the operating system.

Figure 7 shows a breakdown of TreadMarks overhead. We have divided the overhead into three categories: memory management, consistency, and "other". "Memory management" overhead is the time spent at the *user-level* detecting and capturing changes to shared pages. This includes twin and diff creation and diff

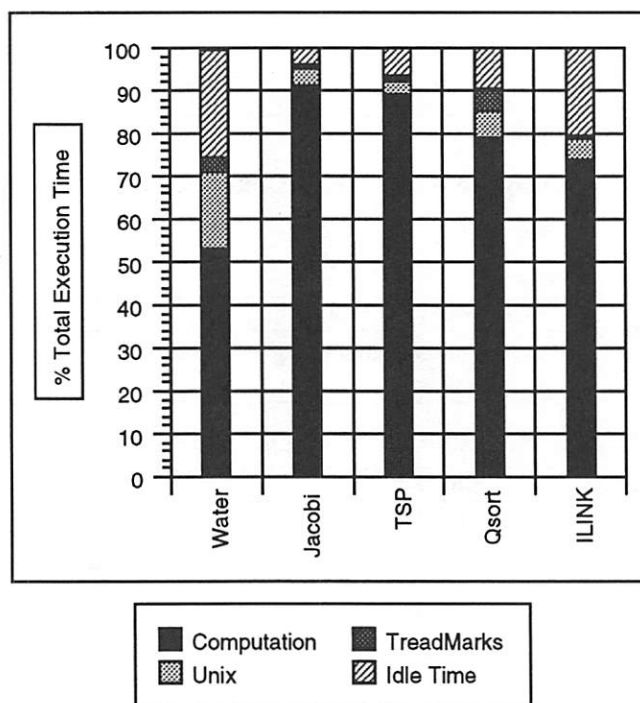


Figure 5 TreadMarks Execution Time Breakdown

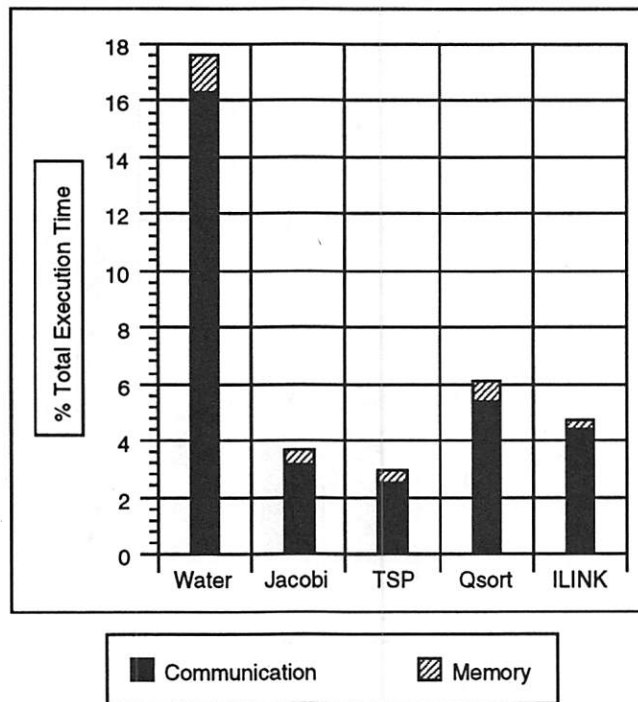


Figure 6 Unix Overhead Breakdown

application. “Consistency” is the time spent propagating and handling consistency information. “Other” consists primarily of time spent handling communication and synchronization. TreadMarks overhead is dominated by the memory management operations. Maintaining the rather complex partial ordering between intervals adds only a small amount to the execution time.

4.6 Effect of Network and Communication Protocol

We ran Water, the application with the highest communication overhead on two other communication substrates: UDP over the ATM network, and UDP over an Ethernet. Figure 8 shows the total 8-processor execution times for all three different communication substrates and a breakdown into computation, Unix overhead, TreadMarks overhead, and idle time.

Overall execution time increases from 15.0 seconds on ATM-AAL3/4 to 17.5 seconds on ATM-UDP and to 27.5 seconds on Ethernet-UDP. Computation time and TreadMarks overhead remain constant, Unix overhead increases slightly, but the idle time increases from 3.9 seconds on AAL3/4 to 5.0 seconds on ATM/UDP, and to 14.4 seconds over the Ethernet. The increase from ATM-AAL3/4 to ATM-UDP is due to increased protocol overhead in processing network packets. For the Ethernet, however, it is largely due to network saturation.

4.7 Summary

TreadMarks achieves good speedups for Jacobi, TSP, Quicksort, and ILINK on the 100 Mbit/sec ATM LAN. For a slightly modified version of the Water program from the Splash benchmark suite, TreadMarks achieved only a moderate speedup, because of the large number of small messages.

The overhead of the DSM is dominated by the communication primitives. Since wire time is negligible on the ATM LAN for our applications, the greatest potential to improve overall performance is reducing

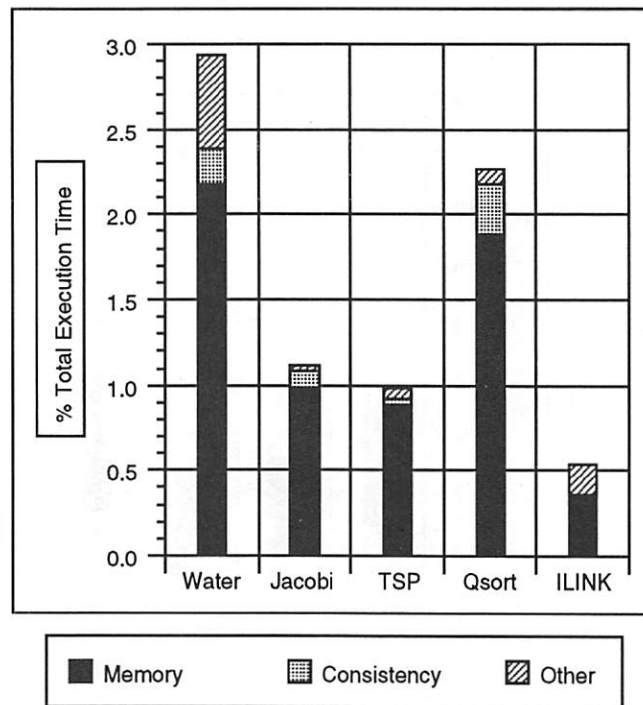


Figure 7 TreadMarks Overhead Breakdown

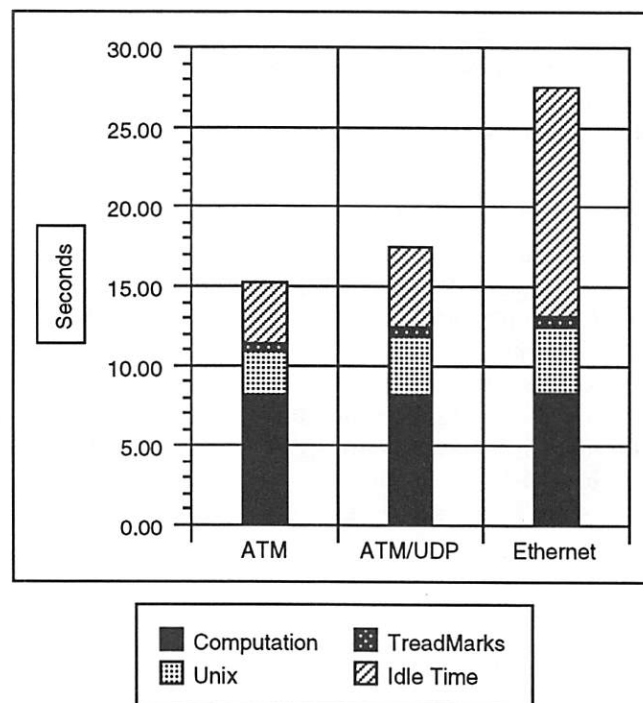


Figure 8 Execution Time for Water

the *software* communication overhead. Although the use of the lightweight AAL3/4 protocol reduces the total send and receive time, these are only a part of the overall communication overhead. Lower-overhead user-level communications interfaces or a kernel-level implementation would improve performance.

A kernel implementation of the memory management would have little effect on overall performance. In the worst case (Water), TreadMarks spent less than 2.2% of its time detecting and capturing changes to shared pages. Most of this time is spent copying the page and constructing the diff. Less than 0.8% of the time is spent in the kernel generating the signal or performing the `mprotect`.

5 Lazy vs. Eager Release Consistency

5.1 Eager Release Consistency: Design and Implementation

We implemented an *eager* version of RC (ERC) to assess the performance differences between ERC and LRC. At the time of a release, ERC creates diffs of modified pages, and distributes each diff to all processors that cache the corresponding page. Our implementation of ERC uses an update protocol. Eager invalidate protocols have been shown to result in inferior performance for DSM systems [14]. We are thus comparing LRC against the best protocol available for ERC. With an eager invalidate protocol, the diffs cause a large number of invalidations, which trigger a large number of access misses. In order to satisfy these access misses, a copy of the entire page must be sent over the network. In contrast, lazy invalidate protocols only move the diffs, because they maintain enough consistency information to reconstruct valid pages from the local (out-of-date) copy and the diffs.

5.2 Performance

Figures 9 to 12 compare the speedups, the message and data rates, and the rate of diff creation between the eager and lazy version of the five applications. In order to arrive at a fair comparison of the message and the data rate, we normalize these quantities by the average execution time of ERC and LRC.

LRC performs better than ERC for Water and Quicksort, because the LRC sends fewer messages and a smaller amount of data. In Water, in particular, ERC sends a large number of updates at each release, because all processors have copies of most of the shared data.

Jacobi performs slightly better under LRC than under ERC. Although communication requirements are similar in both cases, Figure 12 shows that the lazy diff creation of LRC generates 25% fewer diffs than ERC, thereby decreasing the overhead. For ILINK, performance is comparable under both schemes.

For TSP, ERC results in better performance than LRC. TSP is implemented using a branch-and-bound algorithm that uses a *current minimum* to prune searching. The performance on LRC suffers from the fact that TSP is not a *properly labeled* [13] program. Although updates to the current minimum tour length are synchronized, read accesses are not. Since LRC updates cached values only on an *acquire*, a processor may read an old value of the current minimum. The execution remains correct, but the work performed by the processor may be redundant since a better tour has already been found elsewhere. With ERC, this is less likely to occur since ERC updates cached copies of the minimum when the lock protecting the minimum is released. By propagating the bound earlier, ERC reduces the amount of redundant work performed, leading to a better speedup. Adding synchronization around the read accesses would deteriorate performance, given the very large number of such accesses.

6 Related Work

Among the many proposed relaxed memory consistency models, we have chosen release consistency [13], because it requires little or no change to existing shared memory programs. An interesting alternative is *entry consistency* (EC) [4]. EC differs from RC in that it requires all shared data to be explicitly associated with some synchronization variable. On a lock acquisition EC only propagates the shared data associated with that lock. EC, however, requires the programmer to insert additional synchronization in shared memory programs to execute correctly on an EC memory. Typically, RC does not require additional synchronization.

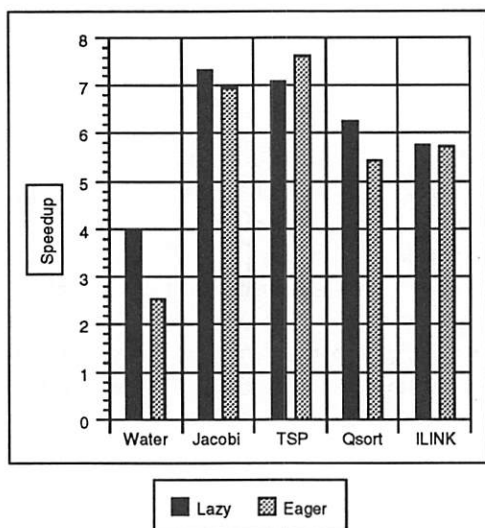


Figure 9 Comparison of Lazy and Eager Speedups

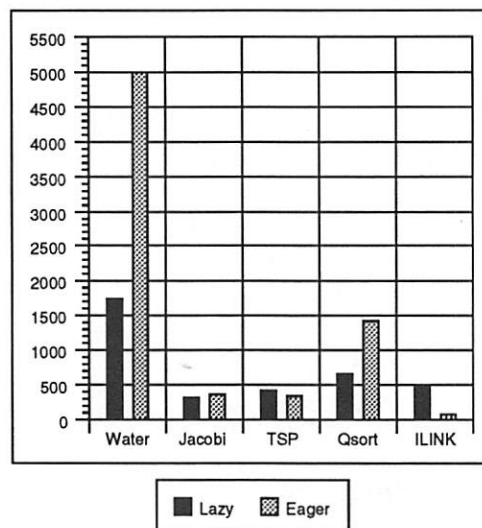


Figure 10 Message Rate (messages/sec)

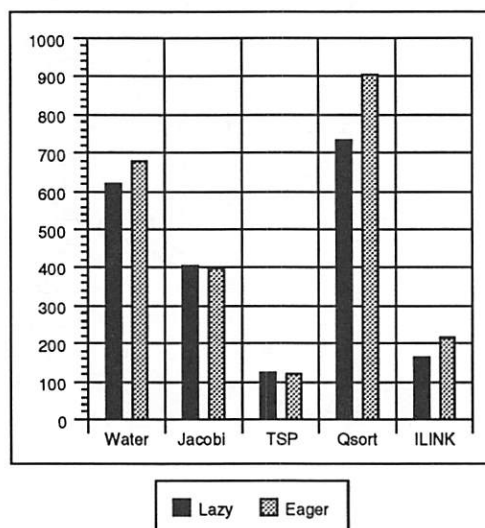


Figure 11 Data Rate (kbytes/sec)

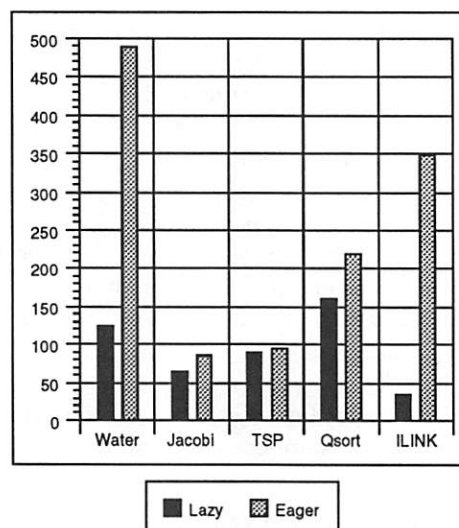


Figure 12 Diff Creation Rate (difs/sec)

In terms of comparisons with other systems, we restrict ourselves to implementations on comparable processor and networking technology. Differences in processor and network speed and their ratio lead to different tradeoffs [9], and makes comparisons with older systems [3, 8, 11, 12, 18, 21] difficult. We have however borrowed from Munin [8] the concept of multiple-writer protocols. Munin also implements eager release consistency, which moves more messages and data than lazy release consistency.

Bryant et al. [7] implemented SSVM (Structured Shared Virtual Memory) on a star network of IBM RS-6000s running Mach 2.5. Two different implementation strategies were followed: one using the Mach external pager interface [24], and one using the Mach exception interface [5]. They report that the latter implementation—which is very similar to ours—is more efficient, because of the inability of Mach's external pager interface to asynchronously update a page in the user's address space. Also, the time to update a page in a user's address space is higher for the external pager interface than for the exception interface (1.2 vs. 0.7 milliseconds) because the need for a `data_request - data_provided` message transaction when using the external pager interface. The overhead of a page fault (without the actual page transfer) is approximately 1 milliseconds, half of which is attributed to process switching overhead in the exception-based implementation. The time to transfer a page (11 milliseconds) dominates all other overheads in the remote page fault time.

Bershad et al. [4] use a different strategy to implement EC in the Midway DSM system, running on DECStation-500/200s connected by an ATM LAN and running Mach 3.0. Instead of relying on the VM system to detect shared memory updates, they modify the compiler to update a software dirty bit. Our results show that, at least in Ultrix and we suspect in Mach as well, the software communication overhead dominates the memory management overhead.

DSVM6K [6] is a sequentially consistent DSM system running on IBM RS/6000s connected by 220-Mbps fiber optic links and a nonblocking crossbar switch. The system is implemented inside the AIX v3 kernel and uses a low-overhead protocol for communication over the fiber optic links (IMCS). A remote page fault takes 1.75 milliseconds when using IMCS, and is estimated to take 3.25 milliseconds when using TCP/IP. The breakdown of the 1.75 milliseconds page fault time is: 1.05 milliseconds for DSVM6K overhead, 0.47 milliseconds for IMCS overhead and 0.23 milliseconds of wire time. Shiva [19] is an implementation of sequentially consistent DSM on an Intel IPSC/2. Shiva is implemented outside the kernel. A remote page fault takes 3.82 milliseconds, and the authors estimate that time could be reduced by 23 percent by a kernel implementation. In comparison, our page fault times are 2.8 milliseconds using AAL3/4. While these numbers are hard to compare because of differences in processor and networking hardware, our results highlight the cost of the software communication overhead. Either an in-kernel implementation or fast out-of-kernel communication interfaces need to be provided in order to build an efficient DSM system.

7 Conclusions

Good performance has been achieved for DSM systems built on various research operating systems. However, in order to use DSM as a platform for parallel computation on clusters of workstations, efficient user-level implementations must be available on commercial operating systems. It is with this goal in mind that we set out to conduct the experiment described in this paper.

We implemented a DSM system at the user level on DECstation-5000/240's connected to a 100-Mbps ATM LAN and a 10-Mbps Ethernet. We focused our implementation efforts on reducing the cost of communication, using techniques such as lazy release consistency, multiple-writer protocols, and lazy diff creation. On the ATM network, we avoided the overhead of UDP/IP by using the low-level AAL3/4 protocol.

On the ATM network, we achieved good speedups for Jacobi, TSP, Quicksort, and ILINK, and moderate speedups for a slightly modified version of Water. Latency and bandwidth limitations reduced the speedups by varying amounts on the Ethernet. We conclude that user-level DSM is a viable technique for parallel computation on clusters of workstations connected by suitable networking technology.

In order to achieve better DSM performance for more fine-grained programs like Water, the software communication overhead needs to be reduced through lower-overhead communication interfaces and implementations.

References

- [1] S. Adve and M. Hill. Weak ordering: A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.
- [2] S. Ahuja, N. Carreiro, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.
- [3] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. A distributed implementation of the shared data-object model. *Distributed Systems and Multiprocessor Workshop*, pages 1–19, 1989.
- [4] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway distributed shared memory system. In *Proceedings of the '93 CompCon Conference*, pages 528–537, February 1993.
- [5] D. Black, D. Golub, R. Rashid, A. Tevanian, and M. Young. The Mach exception handling facility. *SigPlan Notices*, 24(1):45–56, May 1988.
- [6] M.L. Blount and M. Butrico. DSVM6K: Distributed shared virtual memory on the Risc System/6000. In *Proceedings of the '93 CompCon Conference*, pages 491–500, February 1993.
- [7] R. Bryant, P. Carini, H.-Y. Chang, and B. Rosenburg. Supporting structured shared virtual memory under Mach. In *Proceedings of the 2nd Mach Usenix Symposium*, November 1991.
- [8] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.
- [9] S. Dwarkadas, P. Keleher, A.L. Cox, and W. Zwaenepoel. Evaluation of release consistent software distributed shared memory on emerging network technology. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 244–255, May 1993.
- [10] S. Dwarkadas, A. A. Schäffer, R. W. Cottingham Jr., A. L. Cox, P. Keleher, and W. Zwaenepoel. Parallelization of general linkage analysis problems. To appear in *Journal of Human Heredity*, 1993.
- [11] B. Fleisch and G. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 211–223, December 1989.
- [12] A. Forin, J. Barrera, and R. Sanzi. The shared memory server. In *Proceedings of the 1989 Winter Usenix Conference*, pages 229–243, December 1989.
- [13] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.
- [14] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.
- [15] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.
- [16] G.M. Lathrop, J.M. Lalouel, C. Julier, and J. Ott. Strategies for multilocus linkage analysis in humans. *Proceedings of National Academy of Science*, 81:3443–3446, June 1984.
- [17] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.
- [18] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.
- [19] K. Li and R. Schaefer. A hypercube shared virtual memory system. *1989 International Conference on Parallel Processing*, 1:125–131, 1989.

- [20] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52-60, August 1991.
- [21] U. Ramachandran and M.Y.A. Khalidi. An implementation of distributed shared memory. *Software: Practice and Experience*, 21(5):443-464, May 1991.
- [22] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.
- [23] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315-339, December 1990.
- [24] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 63-76, October 1987.

Pete Keleher received the B.S. and M.S. degrees from Rice University in 1986 and 1992 respectively. He is currently a Ph.D student in the Department of Computer Science at Rice University. His research interests include parallel and distributed systems, computer networks and architecture, and parallel processing.

e-mail: pete@cs.rice.edu

Alan Cox received the B.S. degree from Carnegie Mellon University in 1986 and the M.S. and Ph.D. degrees from the University of Rochester in 1988 and 1992, respectively. He is currently on the faculty at Rice University. His research interests include parallel and distributed systems, distributed garbage collection, and multi-computer architectures.

e-mail: alc@cs.rice.edu

Sandhya Dwarkadas received the B.Tech. degree from the Indian Institute of Technology, Madras, India, in 1986, and the M.S. and Ph.D. degrees from Rice University in 1989 and 1993. She is currently a research scientist at Rice University. Her research interests include parallel and distributed systems, parallel computer architecture, parallel computation, simulation methodology, and performance evaluation.

e-mail: sandhya@cs.rice.edu

Willy Zwaenepoel received the B.S. degree from the University of Gent, Belgium, in 1979, and the M.S. and Ph.D. degrees from Stanford University in 1980 and 1984. Since 1984, he has been on the faculty at Rice University. His research interests are in distributed operating systems and in parallel computation. While at Stanford, he worked on the first version of the V kernel, including work on group communication and remote file access performance. At Rice, he has worked on fault tolerance, protocol performance, optimistic computations, and distributed shared memory.

e-mail: willy@cs.rice.edu

Workstation Support for Real-time Multimedia Communication

Olof Hagsand and Peter Sjödin
Swedish Institute of Computer Science

Abstract

We show how multimedia applications with real-time requirements can be supported in a distributed system. A UNIX system has been modified to give soft real-time support. The modifications include deadline-based scheduling, preemption points and prioritized interrupt processing. In addition, a system call interface for real-time application programming has been designed. We justify the modifications by experiments with a simple distributed multimedia delivery system. The experiments are made on an ATM network, where resources are reserved by means of the ST-2 internetworking protocol.

1 Introduction

Modern workstations support multimedia interaction. They are (or can be) equipped with loudspeakers, microphones, video cameras and high-resolution displays. Such devices are different from “traditional” computer I/O-devices since they are *continuous*, which require special treatment by the workstations. Consider as an example of a distributed multimedia application the “netphone”, an application for audio communication between workstations (see Figure 1). Each workstation has two processes, *mike* and *speaker*. The mike process collects sound samples from the microphone, puts them in packets and sends them on the network. The speaker process receives sample packets from the network, unpacks the sound samples and feeds them to the loudspeaker.

The netphone is an application for human communication and therefore has stringent timing requirements. If the delay from microphone to speaker is too long, the round-trip time between the two users will make the netphone annoying to use. If sound samples are not fed to the loudspeaker at regular intervals, the sound will be distorted and the netphone will be difficult, or even impossible, to use. Thus, the netphone has requirements on delay as well as on delay variation. It is hard to define an upper bound on delay for voice communication, 100–250 milliseconds has been suggested [HSS90, And93]. Other applications, such as distributed musical systems, may require delays down to a few milliseconds.

Traditional time-sharing operating systems fail to provide adequate support for real-time multimedia communication. In such systems, the more processes there are, the more seldom each process executes. For many applications, this is acceptable—it just takes longer time for them to fulfill their tasks on machines with heavy load. It is not acceptable for the netphone, however, since the mike and speaker processes need to meet their deadlines even when the load is high.

Our experiences with netphone and similar applications are that when the load increases, they are executed in a way that renders them practically useless. It should be pointed out that the problem with general-purpose workstations is not the hardware performance—on machines with light load, multimedia applications run without problems. Similar experiences have been made, for instance, with the ACME multimedia server on a Sun workstation [GA91].

Real-time operating systems are designed for applications with strict timing requirements, so we could use such a system for the netphone. Other researchers have demonstrated that it is possible to use real-time operating systems, for instance the YARTOS operating system [JSS92], for multimedia communication.

Real-time operating systems are often oriented towards command and control systems. These systems have so called *hard* deadlines, meaning that the cost of missing a deadline is very high. Distributed multimedia have more relaxed timing requirements. For instance, from experiments with two-way video communication, it is reported that 4 video frames out of 1000 can be lost without significant loss in of quality [JSS92]. We think that multimedia capabilities should be integrated in existing workstations, and therefore prefer to study whether current workstations can be modified to better support multimedia.

In this paper, we determine how an existing operating system can be modified to allow multimedia applications to execute undisturbed by concurrent activities. Specifically, we wanted to show how netphone could be made useful in a networking environment. The experiments were made on Sun SPARCstations running SunOS 4.1.1, which, in all aspects relevant for this paper, is equivalent to 4.3BSD UNIX [LMKQ89]. The Sun SPARCstations are connected by an ATM network.

In Section 2, we describe and motivate support for real-time multimedia applications in end-systems and networks. Section 3 describes the necessary modifications of the SunOS kernel. Then, in Section 4, we report and discuss measurements of a simulated multimedia implementation, and Section 5 concludes the paper.

2 Real-time for distributed multimedia systems

A common model of distributed multimedia applications is to regard them as sets of coordinated continuous streams between input and output devices. Although a convenient conceptual model, it is not an adequate description of a multimedia system from a computer system point of view. The main problem is the continuous streams—general-purpose computers are highly asynchronous, and handle data in chunks.

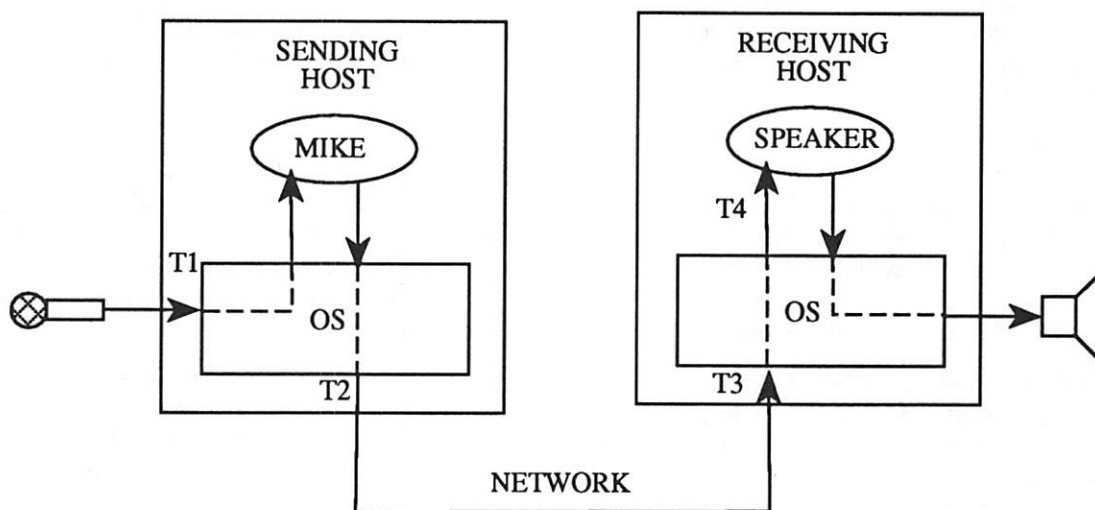


Figure 1: Communication between one microphone and one speaker in the netphone application.

In this paper, we consider simplified multimedia systems (exemplified by netphone) consisting of two hosts: a source and a destination. When data is available on an input device at the source, a process collects the data, processes it, and sends it on the network. At the destination host, the data arrives to a process that feeds it to an output device.

The multimedia application tries to provide a constant delay between the input and output devices (the input-

output delay). The input-output delay is determined by the processing time in the sender and destination processes, the scheduling delay in the source and destination hosts (the time a runnable process is blocked from execution), and the network delay. The application is based on an assumed worst-case input-output delay, and the application delays artificially all data items that can be fed to the output device with a delay shorter than the worst-case. In this way, the multimedia application guarantees a constant input-output delay equal to the assumed worst-case delay, regardless of jitter in the network, for example.

We distinguish between two types of delay: application-specific delay and system delay. The application-specific delay can be caused by copying of data to and from multimedia devices, processing in the sender and destination processes, and so on. The system delay consists of network delay and scheduling delay. Our aim is to reduce the worst-case input-output delay for multimedia applications in general, and therefore we focus on the system delay and do not consider the application-specific delay.

The above is a description of unicast (single destination) applications, but we think it applies to multicast (multiple destinations) applications as well. We assume that the distribution of multicast messages is taken care of entirely by the network, so the hosts and processes in a multicast application can operate as if it was a unicast application. Under this assumption, we do not need to distinguish between unicast and multicast applications.

We use for our experiments a simplified version of the netphone that has a minimal amount of application-specific delay (it sends "dummy" data instead of sound samples from the microphone). Our goal is to limit the system delay to 10 ms. An input-output delay of 100 ms seems to be acceptable for real telephony applications, which leaves 90 ms for application-specific delay. This appears to be realistic, even for CD-quality sound. Video communication has roughly the same delay requirements, so 10 ms scheduling and network delay would be sufficient also for video. However, in our experience, video communication has high application-specific delay, since it requires higher throughput and more processing. In addition, special hardware support is needed to achieve medium-quality video communication.

2.1 Real-Time Support in the Scheduler

We are interested in workstations in a networking environment with many sources of disturbances. Operating systems must process interrupts spawned by other activities on the machine, other processes on the machine execute, networks introduce latencies, etc.

We cannot make any assumptions about when disturbances occur, and we cannot predict exactly when communication events take place in the multimedia system. This has lead us to implement support for event-driven real-time processes. Such processes are triggered by external events, rather than an internal clock. In the netphone, the external events are arrivals of network packets on the destination side, and sound samples on the source side. When an event occurs, the process becomes runnable and needs to perform its task before a *deadline* expires. A runnable real-time process that has not yet met its deadline is *critical*.

2.2 Real-time Support in Networks

In order to maintain an upper bound on network delay, there must be some support for real-time in the network. Ethernet, for example, does not provide this. A packet on an Ethernet can be delayed for an unpredictable amount of time due to other network traffic. We have therefore chosen to experiment with an ATM network (from Fore systems Inc.), which provides practically constant network delay.

ST-2 [Top90] is an experimental internetworking protocol for multimedia applications. It is a connection oriented protocol (point-to-multipoint) where *streams* are constructed from one originator to a set of recipients. A stream is constructed through resource reservations (formulated as flow specifications) that are passed from the sender to the network. ST-2 allows the sender to reserve resources on the ATM network, by passing flow specifications from the application to the resource reserving facilities in the ATM network via the ATM signalling protocol. We have used SICS' implementation of ST-2 [PP91] and interface to ATM [HP93].

3 Modifications of the Operating System

The main effort of our modifications was to implement real-time support in the operating system and design a programming interface for real-time processes. Much of this work was influenced by RT-MACH [TNR90].

3.1 Scheduling

Our intentions were to modify an existing system to incorporate deadline-based scheduling of real-time processes. We chose to implement the earliest deadline first (EDF) scheduling algorithm [LL73]. EDF is well-suited for event-driven aperiodic processes since it does not require real-time processes to be periodic. Essentially, EDF schedules processes by giving the highest priority to the process with the closest deadline.

A critical process runs at such a high priority that it cannot be interrupted by non-critical processes. It runs until it voluntarily yields control of the processor, the deadline expires, or a new critical process appears with a deadline that is closer in time. Our goal is to minimize startup time and the effects of disturbances during execution.

Our method is to make only minor modifications to the existing scheduling code. When a critical process is scheduled for execution by the deadline scheduler, it is put at the head of the highest priority run queue before a context switch is invoked. This guarantees that the critical process will run next. The actual context switch code can therefore be left unmodified. When a critical process yields control of the CPU, the deadline scheduler is invoked to find the next critical process, if any.

3.2 Application Programming Interface

We designed a programming interface for real-time processes based on three new system calls: `rt_select` and `rt_periodic` to launch aperiodic and periodic real-time processes, respectively, and `rt_deadline` to signal that a critical process has met its deadline (and thus turned non-critical).

Our event-driven interface is the `rt_select` system call, which is based on the SunOS 4.1 `select` system call. A process calls `rt_select` to register file descriptors that should trigger critical periods. The process then calls `select`. If an event occurs on one of the file descriptors registered in the `rt_select` system call, the process turns critical. The process stays critical until it either calls `select` again, calls `rt_deadline` to signal that the deadline is met, or misses the deadline. The syntax of `rt_select` is as follows:

```
int rt_select (width, readfds, writefds, exceptfds, deadline, options)
int width;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *deadline;
int options;
```

where `readfds`, `writefds` and `exceptfds` indicate the file descriptors that should trigger critical periods. `Deadline` specifies the maximum time the process may be critical before a deadline expires. The `options` field specifies how the process should be notified of a start of a critical period and deadlines, etc.

The interface also contains a periodic real-time interface, where a periodic process is initiated with a call to `rt_periodic`. The syntax of `rt_periodic` is as follows:

```
int rt_periodic (value, period, deadline, options)
struct timeval *value;
struct timeval *period;
struct timeval *deadline;
int options;
```

where `value` specifies when the first period begins, `period` specifies the time between each start of a critical period and `deadline` specifies the maximum time the critical task may execute after a critical period starts.

We have used the UNIX signal facility to indicate beginning and end of critical periods. A process can then register signal handlers to be invoked at the start of a critical period and a missed deadline, respectively. The process indicates that it has finished its critical period either by calling `rt_deadline`, or by the normal return of the signal handler. Missing a deadline in a soft real-time system is not considered as catastrophic. However, we have chosen to lower the priority, so that a missed deadline will not affect the execution of other critical processes.

3.3 Preemption Points

We found the above changes to be insufficient for supporting multimedia on machines with concurrent activities—it can still take long time before a critical process starts running. This is because context switches cannot occur at any time in SunOS 4.1. In particular, the SunOS 4.1 kernel is not preemptive, which means a process operating in kernel mode (executing a system call) cannot be preempted at all. Some system calls can take long time to execute, and thus block critical processes for long periods of time. We tried to identify such system calls and insert *preemption points* into them. At a preemption point, a (non-critical) process checks whether there is a critical process. If there is, the process yields the processor and thereby preempts itself.

System calls that take long time are typically calls that iterate through a set of objects, and perform some operation on each object. For example, the `exit` system call loops through the memory pages of a process and releases them, one by one. This takes a long time for processes with many pages. Other significant examples are truncation of large files (in the `open` system call), flushing the file system cache (`sync`), copying process memory (`fork`), and loading executable files (`execve`).

Inserting preemption points is a simple method of limiting the blocking time of critical processes. A drawback is that it is often necessary to put preemption points in the bodies of loops. Such preemption points will execute often, and may slow down the system. This illustrates a trade-off between real-time guarantees and execution time; by slowing down the system, we limit the time a critical process can be blocked.

A more general solution is to make the kernel “fully” preemptive, as has been done with SunOS 5.0 [KSZ92]. In a fully preemptive kernel, processes can be preempted at any place. Critical regions in the kernel still need to be protected, so fully preemptive kernels need mechanisms for mutual exclusions (e.g. locks and semaphores). This means that whenever a process wishes to enter a critical region, it must first execute some code to ensure exclusive access to the region. Thus, fully preemptive kernels also trade execution time for real-time properties.

3.4 Interrupt Activity

We found interrupt handling to be another source of disturbances for real-time processes. Interrupt handling routines can be thought of as high-priority processes. Once an interrupt occurs, the interrupt handler starts executing and preempts any process that is running.

In SunOS 4.1, interrupt processing may be done at different priority levels. The hardware interrupt handler does the urgent interrupt processing, and then schedules the software interrupt process, “softint”, to do interrupt processing at lower priority. For example, a network driver working at a high priority level enqueues incoming packets and re-initializes network interface hardware, but schedules softint to do the protocol processing for the incoming packets.

The softint processing is done at interrupt level, which means that it preempts any process that is running. Much of the softint processing, in our opinion, is less urgent than critical processes, and can be postponed until there are no critical processes. There is, however, softint processing that need to be done immediately, for instance processing of incoming data for real-time applications (such as incoming audio packets for the netphone).

As an experiment we chose to assign a lower priority to IP packets arriving from the network interface than to ST-2 packets. If there are critical processes, processing of IP packets is postponed. This intermediate

solution is based on the assumption that ST-2 packets carry real-time data, whereas IP packets do not. Although this is true for the experiments we have conducted, it is not a sound assumption in general. One would rather prioritize real-time data independently of the protocol that carries it over the network.

The question is if it is possible to discern at interrupt level whether or not incoming data is real-time data? It seems possible to use `rt_select` for this: In the networking case, a file descriptor has an associated protocol control block with an *end-point identifier*. For TCP the destination port could be used as end-point identifier, and for ST-2 the hop identifier.

The idea here is that it is easy to extract the corresponding end-point identifier from a packet's header. For each incoming packet the header is examined, and if it matches a protocol control block that is associated to a "real-time" file descriptor, softint is called immediately, otherwise it is postponed.

4 Experiments

In this section, we describe some experiments made in order to verify that our real-time modifications improve real-time performance.

Two SUN 4/65 (SPARCstation 1+) are connected by a local area ATM 100 Mb/s network, consisting of two 8×8 ATM switches from Fore Systems Inc. The workstations are also connected by an Ethernet. The experimental application is a simplified version of netphone (see Figure 1) where all audio device dependencies are removed. In this version, the mike process sets up an ST-2 connection, generates data and sends packets at a fixed rate. Each packet is timestamped four times (see Figure 1):

- T1. When a triggering event (i.e., a timeout) occurs at the sender.
- T2. When the packet is sent on the network.
- T3. When the packet is received from the network in the receiving machine.
- T4. When packet is delivered to the speaker process.

The timestamps are saved and then analyzed off-line, while the data itself is dropped at the destination.

The two workstations are synchronized through NTP [Mil92], but the synchronization is not accurate enough to give an absolute value of the network delay (i.e., the difference between T3 and T2). We therefore assume a fixed network delay of 1 millisecond, which is a slight exaggeration.

In the experiments, we try to use bit-rates typical for multimedia applications. We use two bit-rates, 1.2 Mb/s and 16 Mb/s. 1.2 Mb/s is to mimic audio and compressed medium-quality video [ISO92]. Here, 1500-byte packets are sent at a rate of 100 Hz. 16 Mb/s represents uncompressed video and low quality compressed HDTV. For 16 Mb/s, we have to use a SPARCstation 2 as receiver, an increased rate of 500 Hz, and packet size of 4K bytes. All experiments run over a period of 60 seconds. In the 1.2 Mb/s case, we make a comparative study of transfer over Ethernet.

Our goal is to minimize the effects of disturbances on multimedia communication. It is therefore necessary to apply disturbances to evaluate the real-time modifications. We apply a competing activity on both machines that is both CPU and I/O intensive. This activity performs a compilation, using `make`, of a C program where the sources are stored remotely on an NFS file system. Therefore, there is I/O activity over an Ethernet and a local disc. In addition, all experiments are made under normal working conditions, so there is other traffic on the networks, ordinary background activities on the workstations, and so on.

Table 1 shows the *end-to-end delay* ($T4 - T1$) for six different experiments. In the table, the minimum, maximum and average delay are given, as well as delay variance. There are four experiment parameters in the table:

RT A "+" indicates that all real-time kernel modifications were turned on at both sender and receiver. A "-" indicates an unmodified SunOS kernel. In the latter case, the mike process is triggered by an interval timer, and speaker blocks in a read loop.

Experiment number	Options				Min	Max	Mean	Variance
	RT	Net	PP	Rate	[10^{-3} s]	[10^{-3} s]	[10^{-3} s]	[10^{-6} s ²]
1	+	ATM	+	1.2	2.5	10.0	3.3	0.3
2	-	Ether	-	1.2	2.7	445.0	16.8	2224.2
3	-	ATM	-	1.2	2.4	188.5	5.3	136.8
4	+	Ether	+	1.2	2.6	39.9	3.3	1.7
5	+	ATM	-	1.2	2.5	103.2	3.3	5.3
6	+	ATM	+	16.0	2.1	26.5	3.1	3.8

Table 1: End-to-end delay ($T_4 - T_1$).

Net The network is either ATM or Ethernet.

PP A “+” indicates that the preemption points in the kernel are enabled. This option can only be set if the real-time kernel is used.

Rate The transfer rate is given in Mb/s.

Experiment number 1, over ATM and with kernel real-time modifications, shows that we barely reached our goal to keep input-output delay below 10 milliseconds. Experiment 6, over Ethernet and without kernel real-time support, is the “worst” possible and shows much higher maximum and average delay.

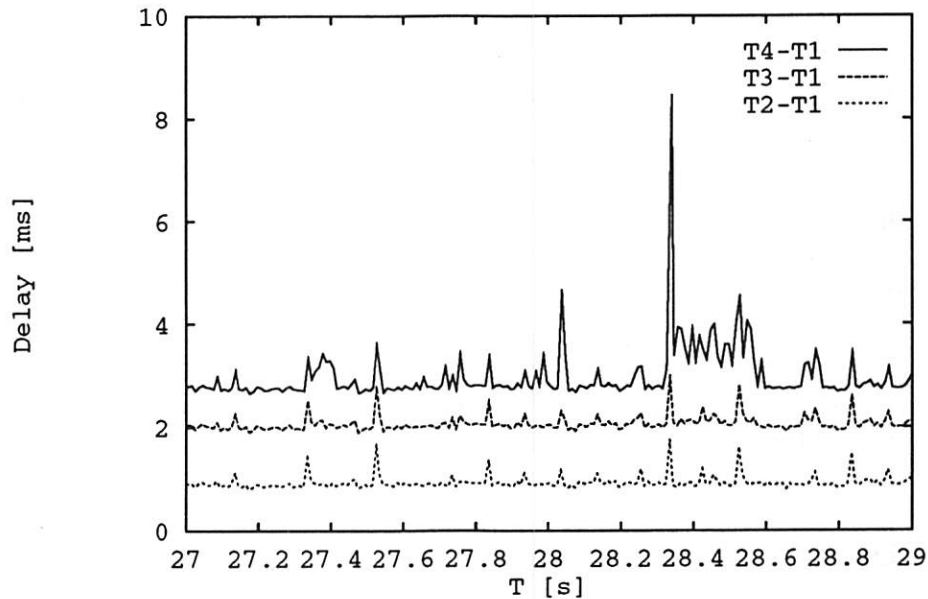


Figure 2: End-to-end delay on ATM with kernel real-time modifications (experiment 1).

Figure 2 and Figure 3 show delay for packets sent during two seconds of experiment 1 and 2, respectively. In these experiments, packets are sent at ten millisecond intervals. The x-axis (in seconds) is the relative time T from the start of the experiment, ranging at most from 0 to 60 seconds. We picked different time intervals from experiment 1 and 2, and therefore the range on the x-axes differ. The y-axis is the difference in time between a timestamp and the first timestamp, T_1 . That is, for each packet we plot, versus T , the difference in time (the delay) between the time of the triggering event at the sender (T_1) and the time when

the packet enters the network (T2), leaves the network (T3) and arrives at the speaker process (T4). In both figures, we can see that many packets are delivered with low delay. Some packets are delayed more due to disturbances, represented as spikes in the diagrams. Figure 3 shows that disturbances occur at all places in the system: at the sender, in the network, and at the receiver.

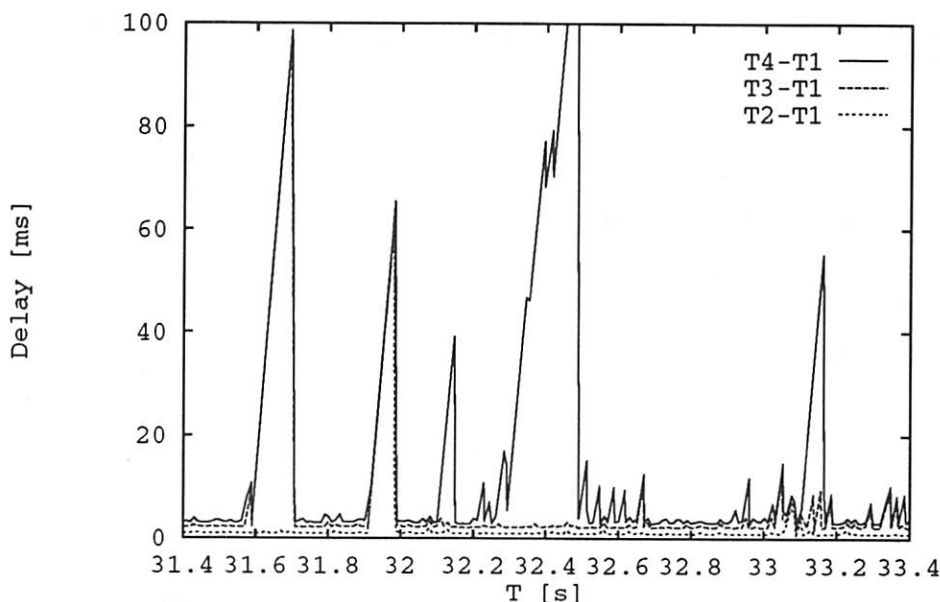


Figure 3: End-to-end delay on Ethernet without kernel real-time modifications (experiment 2).

The third and fourth experiments are included for studying the individual effects of using ATM and our kernel modifications. We see that the maximum delay gets significantly longer both if we use ATM but remove the kernel modifications (experiment 3), and if we keep the kernel modifications but switch to Ethernet (experiment 4). From this we conclude that it is necessary with real-time support in both network and operating system—if only one of them supports real-time, we lose end-to-end real-time properties.

The fifth experiment is made with preemption points turned off. Comparing this with experiment 1, in which preemption points are on, shows that preemption points significantly improve worst-case delay.

In experiment 6, a faster CPU is used as a receiver. Still, the CPU is not fast enough to hinder the buffer buildup which occurred between the socket layer and the speaker process, which we think is the explanation for the increase in maximum delay.

4.1 Prioritized Interrupt Processing

Experiment number	Options	Min [10 ⁻³ s]	Max [10 ⁻³ s]	Mean [10 ⁻³ s]	Variance [10 ⁻⁶ s ²]
	Postpone IP				
7	+	2.1	7.3	3.0	0.3
8	-	2.1	84.5	5.8	42.1

Table 2: End-to-end delay (T4 - T1) with and without prioritized interrupts.

The disturbances are not enough to show effects on postponing IP input processing, as described in Section 3.4. To show such effects, we generate intensive IP traffic from an external source to the receiving host.

Table 2 shows the end-to-end delay for the two cases. As can be seen from the table, maximum delay increases from 7.3 to 84.5 milliseconds if IP input processing is not postponed. Even though this represents an extreme background load, we have shown that IP input processing can disturb critical processes significantly.

5 Conclusions

We have demonstrated that it is possible to modify a standard UNIX (SunOS 4.1) operating system to provide adequate support for multimedia communication. We have modified the scheduler, added three system calls, made the kernel (slightly) preemptive by using preemption points, and modified interrupt handling. We have verified the effect of the changes by measuring the netphone application on an ATM network, both with the modified and the original kernel. The result is encouraging—with our modifications, we limit end-to-end delay to less than 10 milliseconds, and the netphone runs smoothly even on machines with heavy load.

We conclude from the experiments that in order to get real-time communication from end to end, one must have real-time support in the operating system as well as in the network. When we used an operating system without our kernel real-time modifications, or switched from ATM to Ethernet, the worst-case delay increased several times, giving an end-to-end delay much above the maximal 10 milliseconds.

We have suggested a way of improved interrupt processing in the presence of critical processes. As future work, we plan to implement and analyze this improvement. Furthermore, there are other operating systems available with real-time support, and we would like to study how well they support an application such as the netphone.

References

- [And93] D. P. Anderson. Metascheduling for continuous media. *ACM Transactions on Computer Systems*, 11(3):226–252, August 1993.
- [GA91] R. Govindan and D. P. Anderson. Scheduling and IPC mechanisms for continuous media. In *Proceedings of ACM Symposium on Operating Systems Principles, ACM Operating Systems Review*, volume 25, pages 68–80, October 1991.
- [HP93] O. Hagsand and S. Pink. ATM as a link in an ST-2 internet. In *Fourth International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '93)*, Lancaster, November 1993.
- [HSS90] D. B. Hehmann, M. G. Salmony, and H. J. Stüttgen. Transport services for multimedia applications on broadband networks. *Computer communications*, 13(4):197–203, May 1990.
- [ISO92] ISO/IEC. Information technology — coding of moving pictures and associated audio for digital storage media up to 1,5 Mbit/s, 1992. Draft International Standard ISO/IEC DIS 11172.
- [JSS92] K. Jeffay, D. L. Stone, and F. Donelson Smith. Kernel support for live digital audio and video. *Computer communication*, 15(6):388–395, July/August 1992.
- [KSZ92] S. Khanna, M. Sebrée, and J. Zolnovsky. Realtime scheduling in SunOS 5.0. In *Proceedings of the USENIX Winter Conference*, pages 375–390, 1992.
- [LL73] C.L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.
- [LMKQ89] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, May 1989.

- [Mil92] D.L. Mills. Network time protocol (version 3): Specification, implementation, and analysis. *Network Working Group Request for Comments 1305 (RFC 1305)*, March 1992.
- [NYM91] J. Nakajima, M. Yazaki, and H. Matsumoto. Multimedia/realtime extensions for the Mach operating system. In *Proceedings of the Summer 1991 USENIX Conference*, pages 183–198, Nashville, Tennessee, June 1991.
- [PP91] C. Partridge and S. Pink. An implementation of the Revised Internet Stream Protocol (ST-2). In *Proceedings of the 3rd MultiG workshop*, Stockholm, Dec. 1991.
- [TNR90] H. Tokuda, T. Nakajima, and P. Rao. Real-time Mach: Towards a predictable real-time system. In *Proceedings of USENIX Mach Workshop*, October 1990.
- [Top90] C. Topolcic. Experimental Internet Stream Protocol, version 2 (ST-II). *Network Working Group Request for Comments 1190 (RFC 1190)*, Oct. 1990.

Acknowledgements

We wish to thank Bengt Ahlgren and Kjersti Moldeklev for suggestions and comments on this paper, and for patiently providing “normal background activity” on our experiment machines. We also want to thank Fredrik Orava and Stephen Pink for their constructive comments on this paper.

Biographies

Olof Hagsand is a researcher at the High Performance Communication Systems group and the Distributed Collaborative Environments group at the Swedish Institute of Computer Science. Present research interests include high speed networking and distributed virtual reality. He received his MSc degree in engineering physics from Uppsala University in 1986, and is presently a doctoral student at the Department of Teleinformatics at the Royal Institute of Technology in Stockholm. His e-mail address is olof@sics.se.

Peter Sjödin is a researcher in the High Performance Communication Systems group at Swedish Institute of Computer Science. His current research interests include protocols and architectures for high-speed networks, and real-time communication systems. He received his PhD degree in Computer Science from Uppsala University in March 1992. His e-mail address is peter@sics.se.

Experience and Results from the Implementation of an ATM Socket Family

Richard Black and Simon Crosby
University of Cambridge Computer Laboratory
Pembroke Street, Cambridge, CB2 3QG, U.K.

Abstract

This paper describes the implementation of an ATM protocol stack as a protocol family within a 4.3 BSD derived Unix. A novel approach to the implementation of the management and control functions for the ATM protocol stack has been adopted. The data path is implemented within the kernel but all control and management functions are implemented by a user space daemon. An encapsulation of IP on the ATM protocol is provided by means of a logical IP interface. The mapping of IP addresses to ATM addresses is performed by the user space daemon.

1 Introduction

The Cambridge ATM environment is heterogeneous, and includes networks such as the Cambridge Fast Ring (CFR) [8], Cambridge Backbone Ring (CBN) [6], and Fairisle [2]. The CFR and CBN are respectively 50Mb/s and 500Mb/s slotted rings which predate current ATM standards and use a 36 octet cell size, whereas Fairisle is a switch-based ATM network with a standard B-ISDN and ATM Forum 53 octet cell size. To permit easy interconnection of ATM networks and services the ATM protocol is also carried across the Ethernet in the form of "Fat Cells", in which an Ethernet frame is filled with ATM cells in a format which enables rapid cell forwarding at Ethernet-ATM gateways. The software environment is also heterogeneous and supports Unix and experimental micro-kernels. To this environment will be added standards compliant ATM equipment purchased from third parties, and in the near future the network will be connected to the national wide area ATM network provided as part of Super-JANET.

To facilitate the management of this complex system amid constantly changing protocol standards, and to permit both experimental and service use of the network, we decided to implement the management and control functions for the ATM network in a single body of code which could be ported to all of our operating platforms, including 64 bit architectures. This was accomplished by implementing the code as a user space daemon. This paper describes the features implemented in the 4.3 BSD based Ultrix kernel to support the management and control functions in user space, and discusses the design and implementation of the user space manager.

The management code can be extended to accommodate emerging international standards for ATM networks, such as the ATM Forum signalling protocol and the IETF suggested standard for address resolution for IP on ATM [10]. Such extensions can be carried out without modifying the kernel, as all management and control functions exist in user space.

Problems resulting from the asynchronous nature of the control interface are described. We identify several shortcomings of the BSD socket interface for the ATM protocol domain, including the difficulty of describing parameters required for the ATM protocol such as Quality of Service (QoS) specifications and adaptation layer requirements.

2 The ATM protocol

In the ATM protocol domain the socket interface provides an application with direct access to an ATM virtual connection. The ATM protocol is offered as a new address family, `AF_ATM`. Each Protocol Data

Unit (PDU) to be transmitted over the connection is handed directly to the ATM adaptation layer for segmentation into ATM cells and subsequent transmission. Similarly, received PDUs are handed up to the socket layer by the adaptation layer on completion of reassembly. Draft standards for ATM UNI signalling [3] provide a mechanism for the selection of the ATM adaptation layer (AAL) and Quality of Service (QoS) to be used for an ATM connection. However, there is no mechanism in the socket interface to permit an application to specify these parameters to the `connect()` system call in a clean manner. Consequently, for this implementation all ATM connections make use of AAL5, and a default "best-effort" QoS specification is used.

At the ATM layer each connection is represented by an *association* record, and each socket is associated with a single association. In the case where the connection is local, no association record is required and each of the two sockets corresponding to the endpoints of the connection contains a reference to the other. This is similar to the local connection case in TCP. During connection setup the user space manager instructs the ATM protocol code in the kernel to build the data path for the socket. When the connection is complete the protocol control block (PCB) associated with the socket contains a reference to the ATM association record for the connection. Final authority as to the state of a connection lies with the manager. It can unilaterally decide to terminate a connection at any time.

Additionally the ATM code provides logical interfaces to the IP code within the system and will set up tunnels over the ATM network to carry IP traffic to its destination. In this case the upper layer for the ATM protocol is a tunnelling / logical interface engine rather than the socket code. Figure 1 shows a diagrammatic overview of the system components.

3 Communication with the Manager

To permit the ATM connection manager to be implemented in user space, a mechanism was required to permit communication between the kernel socket layer and the manager, and between the ATM layer and the manager. Two possible solutions were identified, namely the provision of a special device in `/dev` or a "magic" control socket type. We chose to implement the latter option, adding a socket type `SOCK_RAW` to `AF_ATM` which can have only one instance per machine. The reasons for this choice are:

- For concurrency reasons it was preferable for the manager to interface directly with the networking code rather than via the file system. This would also make implementations on kernel threads, such as on OSF/1, easier. This is particularly important because most of the information for the manager is generated as a result of calls from the socket layer.
- The user space manager is responsible for the transmission and reception of ATM signalling messages. Since these must be presented to the ATM layer in the form of queues of `mbufs` and the mechanism for the encapsulation of data in `mbufs` is already provided by the socket layer, it is preferable to transmit and receive signalling via the socket interface rather than via the file system. The fundamentally asynchronous nature of interaction between the kernel and the daemon maps well onto the asynchronous nature of socket communication.

The control socket is used to exchange three types of information: socket layer requests, ATM layer requests and ATM signalling. Socket layer requests to the manager include, for example, requests to `bind()` an ATM address to a socket, `connect()` a socket to a destination ATM address, and `close()` a connection. Responses from the manager indicate the status of outgoing connections and include notification of incoming connection requests. The manager uses the control socket to issue commands to the ATM layer to build an association for each connection, provide it with a virtual circuit identifier (VCI), and monitor its status. In the control plane the manager transmits and receives ATM signalling messages via the control socket. Implementation of the ATM control plane in user space has the advantage that modification of the signalling protocol to comply with the latest ATM signalling standards involves only the user space daemon and not the kernel ATM layer code. In addition, development can be aided by the use of programming tools such as debuggers.

Messages exchanged between the kernel and the manager are normally in the form of fixed length control messages. Some of these message blocks, however, may be followed by ATM signalling messages for

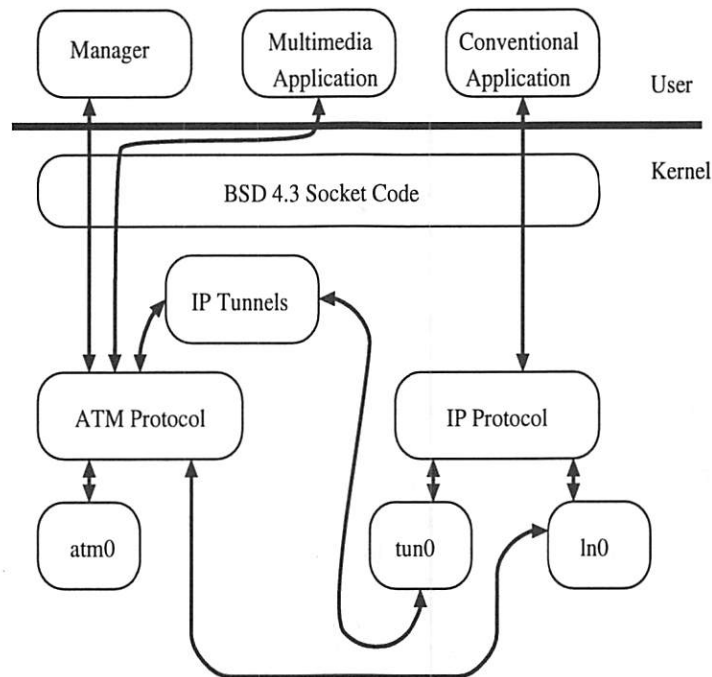


Figure 1: Overview of System Components

transmission or which have been received.

A few management operations cannot be performed asynchronously and are transferred using the `ioctl()` system call. In this case the kernel can synchronously modify the block to return the required response.

4 Kernel Modifications

Two key issues were identified which dictated the design of the kernel code. First, although the user space manager is a part of the operating system, its failures should be localised. If the manager is not running, or crashes, the kernel code should remain consistent and stable. Existing sockets should be closed and additional requests at the socket layer for `AF_ATM` should be denied. Malfunctions on the part of the manager should be incapable of crashing the kernel.

Secondly, both the socket level code and the manager will asynchronously issue requests pertaining to sockets in the ATM protocol domain. Because they operate concurrently it is impossible to ensure that both have a consistent view of the state of all ATM connections at any one time. Special measures must be taken to ensure that errors resulting from this inconsistency are avoided.

To solve these two problems the kernel protocol code keeps in the PCB for each socket its own notion of the state of the socket¹. For example the protocol code will call `soisconnecting()` and even `soisconnected()` to cause the socket code to perform the required operations even when the protocol is not necessarily either connecting or connected. Likewise the PCB continues to exist even when the socket code detaches the socket until the manager has acknowledged that it has finished dealing with the signalling for that PCB.

The kernel code and the connection manager identify individual ATM connections by means of their associated PCBs. To the manager the PCB is an opaque identifier which uniquely identifies an ATM connection. Within the kernel PCBs are kept in an open hash table which is used to verify every PCB reference passed to the kernel by the manager. Each PCB contains a pointer to a set of methods for dealing with the higher level protocol. It also contains methods which implement the transmission path and permit the socket layer

¹or IP tunnel

to destroy the data path, as well as an opaque value which, in the case of a local connection, is a pointer to the PCB for the peer socket, and in the case of a remote connection, is a pointer to the association structure at the ATM layer.

The association structure contains various control fields such as the VCI and a pointer to the receive routine. For normal connections the receive routine performs AAL processing on the received cells before passing up AAL Service Data Units (SDUs) to the socket layer. For each ATM network interface a special association exists which is used for meta-signalling. For these associations the receive routine passes the entire payload of each meta-signalling cell up to the socket layer for forwarding to the manager via the control socket. The manager transmits meta-signalling via the control socket.

5 Kernel Implementation Problems

In the following sections we address several of the implementation problems which were encountered, some of which resulted directly from the design choices described above.

5.1 Socket Addresses

The BSD 4.3 socket code requires the implementation of the `bind()`, `getsockname()`, and `getpeername()` system calls for all protocols to be non-blocking. Many applications perform a `getsockname()` immediately after a `bind()` in order to publish or export their service address. This unfortunately means that the implementation of the ATM service address allocator must remain within the kernel. The addresses allocated are reported to the user space manager. Whilst it would have been preferable to locate this function with the rest of the control plane functions in the manager, this restriction is a minor inconvenience.

5.2 Interrupt levels

BSD 4.3 is structured such that all interaction with the socket level code must be performed while operating at `splnet` whereas device interrupts and the internals of the `mbuf` system operate at `splimp`. BSD 4.3 assumes that there will always be a queue structure between the device driver and the protocol handler. For this reason the ATM device driver cannot manipulate the association data structures, for example during reassembly. Instead it enqueues complete blocks of data on a per VCI basis for the ATM protocol layer. The ATM protocol layer runs at `splnet` and removes blocks from the queue asynchronously, adding them to the receive buffer on the appropriate socket. This additional layering represents a significant bottleneck in the protocol stack and significantly slows down the implementation.

5.3 Accepting Semantics

The ATM protocol implemented on the Cambridge networks, MSNA [12], allows the passive end of a connection to consider both the address of the peer and the quality of service requested before deciding to accept the connection. This is directly at odds with the BSD 4.3 semantics which require that a connection be declared "soisconnected" before the receiver is informed of the connection. In our implementation an incoming connection request is forwarded to the manager which in turn presents the connect request to the kernel ATM layer. On receipt of the connect request the ATM layer declares to the socket layer that the connection is complete (soisconnected), and prevents subsequent transmission on the socket associated with the connection by manipulation of the socket transmit buffer. It then instructs the manager to accept the connection. Once the manager has accepted the connection and built the data path, it informs the ATM layer that the connection is complete and transmission and reception are enabled.

Although a Unix server is still prevented from vetting its peer before accepting a connection, our implementation preserves the expected ATM protocol semantics. The inability of the socket interface to permit an application to vet its peer and to accept connections conditional on the availability of sufficient resources to meet their QoS requirements should be addressed to provide the full flexibility of ATM to applications.

6 IP Logical Interface

Within the kernel a logical IP interface is provided which makes use of ATM connections as “tunnels” – each tunnel is a logical link in the IP network. The IP tunnelling code is responsible for generating requests for the establishment of ATM connections and for the transfer of IP datagrams over these connections. It makes direct use of the protocol code and provides the PCB for each of its tunnels with methods which permit it to monitor the state of the connection.

The tunnelling code sets up an ATM connection to every active IP destination, which are torn down after a period of inactivity. It would be preferable to use one ATM connection for each TCP connection or UDP session, however this would require extensive modification of the BSD 4.3 IP code². An alternative, which has not yet been implemented, would be to identify the higher layer TCP and UDP associations within the ATM driver in a similar manner to that used in SLIP header compression [11].

6.1 IP Address Resolution

To set up an ATM connection for tunnelling IP packets, the IP tunnelling code must somehow resolve the IP address to form an ATM address. It was a particular choice of our design that this mapping would be performed by the manager, and so we permit `AF_INET` addresses to be specified as the destination for an ATM connection. The manager performs the necessary mapping.

Address mapping schemes used previously in the implementation of IP over X.25 use either a fixed algorithm [1] (which requires a private X.25 network), or a manually administered configuration file read at initialisation time such as [13]. In contrast in the scheme we have adopted, any mapping is possible and the management daemon is free to solicit or receive updates or enhancements to its knowledge at any time. For example at boot time the manager may know a single IP to ATM address mapping for its default router. The router could then inform it of optimisations that could be made based on traffic analysis.

6.2 IP MTUs

Recent debate in the internet community has addressed the problem of choosing a suitable Maximum Transfer Unit (MTU) for the encapsulation of IP on ATM. The MTU determines the maximum size of an IP datagram which will be transmitted on the ATM network, and its choice is influenced by several factors. In order to exploit the high bandwidth of the ATM network the MTU should be as large as possible, for example 64KB. On the other hand, some routers, switches and hosts may be unable to support such a large MTU, and fragmentation of IP datagrams into smaller units will result. In addition, if a single ATM connection is used to carry all IP traffic between two hosts then performance of higher layer protocols may be degraded if the MTU is too large because of the delay in processing each large packet as a single unit. If one ATM connection is used per TCP connection then this can be avoided. In a LAN environment a single IP datagram is likely to arrive as a large burst of ATM cells. Care should be taken to ensure that buffer space in the receive interface does not overflow, taking particular account of the interrupt response time of the system under realistic loads.

Clearly some mechanism for the negotiation of the MTU per ATM connection should be provided to permit individual hosts to achieve maximum performance from the ATM network. Our ATM signalling protocol permits the negotiation of the maximum sizes of the encapsulated forward and return PDUs, thereby permitting the MTU to be negotiated per tunnel. In our current implementation we provide four logical IP interfaces, each with a different MTU. This will permit us to experiment with different MTU sizes without modifying the generic IP code. The IP routing tables on each host are configured to use the IP interface with the best MTU for the attached ATM network.

7 Design of the User Space Manager

The user space manager implements the control and management planes for the ATM protocol. It is responsible for ATM network signalling and communicates with the ATM protocol family code in the kernel. It also performs mapping of IP to ATM addresses for the IP tunnelling code.

²This approach has been followed in our local micro-kernel IP implementation

At the socket layer requests issued to the manager include `bind()`, `connect()`, `listen()`, `accept()` and `close()`. Each request specifies a unique identifier (the PCB pointer) which identifies the ATM connection to which it refers. Requests are received asynchronously on the control socket. For each call reference the manager maintains a record of the state of the connection, the association fields such as the VCI, and ATM addressing information. The manager issues requests to the protocol layer to process incoming connection requests and to inform the socket layer of the state of each connection. The manager is responsible for building the data path for each connection at the ATM layer. It issues requests via the control socket to build associations, assign VCIs and monitor the status of each connection. Signalling traffic is transmitted on the appropriate ATM interface by the manager to request the setup of connections on behalf of the protocol layer.

Since the control socket delivers requests asynchronously and since the manager needs to be able to process several requests (from all layers) simultaneously, it is implemented as a multi-threaded process. The implementation uses the POSIX Pthreads package [9] to implement several threads of control to manage the following concurrent tasks:

- receipt and transmission of socket layer requests,
- management of the ATM layer and its associated network interfaces,
- receipt and transmission of ATM signalling, and
- generation of timeouts and performing consistency checks.

These independent threads are scheduled by the Pthreads package, which also implements concurrency control primitives for shared data. As the performance measurements in section 8 show, the concurrency control primitives have a high associated cost, and result in some inefficiency.

An important consideration when designing the manager was the potential inconsistency between the manager and socket layer views of the state of each connection. For example, an application could issue a `listen()` and then immediately `close()` the socket. Due to the asynchronous nature of the control socket, the manager might attempt to complete an incoming connection to the listening socket after the application had already issued the `close()`. To prevent this, the kernel code checks all instructions issued by the manager to ensure that they pertain to valid PCBs. Similarly, the manager will destroy any socket for which the kernel issues an invalid request. Final authority as to the state of a connection lies with the manager.

8 Performance

In this section we present performance measurements for the implementation of the ATM protocol. The performance of the control path was investigated by measuring the time taken to set up connections over the ATM network and the Ethernet. In addition, the manager was profiled to determine which parts of the code were critical in determining its performance.

8.1 Data Path

The implementation of the data path in the kernel is crucial to the performance of the ATM protocol. Comparison of the performance achievable with that for the micro-kernel demonstrate that it is limited by the current structure of the BSD 4.3 protocol stack. Nevertheless, high data transfer rates can be achieved. Performance measurements using a very simple cell based interface with little buffering [7] indicate that a raw ATM throughput of 20Mb/s can be achieved between two DS5000/25 hosts. Using an MTU of 2.5 KB a throughput of 15Mb/s for a TCP connection over IP on ATM was achieved.

8.2 Control Path

The performance of the control path is affected by several factors: scheduling of the manager as a user space process, concurrency within the manager, and communication between the manager and the kernel. Table 1 gives typical connection setup and teardown times over the Ethernet and a direct ATM link between two lightly loaded DS5000/25 hosts running the kernel and user space manager, and for local IPC on a single

host. Measurements are also given when the hosts are connected through an ATM switch. The performance measurements were made using `etp` [4].

Network	Setup (ms)	Teardown (ms)
Ethernet	21.7	8.4
ATM	20.4	8.1
Local	21.8	3.2
ATM (switched)	42.1	8.7

Table 1: Connection setup and teardown times

The measurements indicate that our implementation performs worse than an earlier implementation of a (simpler) version of the control plane within the Unix kernel, for which average connection setup times of 3.5 ms were measured between two Unix hosts over the Ethernet during periods of low network activity [5]. Although it would be unreasonable to expect our implementation to perform as well as a more mature, though less functional, kernel implementation, we might have hoped for better results. The results can be only partially explained by competition between the client and connection manager (and in the local case, the server), all of which were running at the same, default, user priority level. To determine why connection setup took longer than expected, we profiled the connection manager.

The profile results showed that the manager typically spends just over half of its active time (i.e. time not blocked in the `select()` call) executing code in the Pthreads library. In all of our measurements, the 10 most frequently called functions were in the Pthreads library. Most of these functions are concerned with thread context switching and concurrency control. Table 2 lists these functions, their associated frequencies, and the percentage of the total active time of the manager for which they account, for a typical profile during which 20 connections were set up. Clearly the concurrency control primitives have a high cost. The manager could be made more efficient by implementing it as a single threaded process. On an operating system platform which supports lightweight threads, such as OSF/1, it would be hoped that the multi-threaded design of the manager would not impact its performance, and the connection setup time would be proportionately reduced. We believe that this, in conjunction with some further optimisations identified, would enable a reduction of approximately 50 % in the connection setup time, reducing it to the order of 10 ms.

Function	No. of calls	% CPU time
<code>exc_push_ctx</code>	3037	6.411
<code>_setjmp</code>	3037	2.046
<code>exc_pop_ctx</code>	3019	0.716
<code>cma_int_mutex_unblock</code>	948	0.471
<code>cma_int_mutex_block</code>	948	0.426
<code>cma_init</code>	922	0.178
<code>pthread_mutex_unlock</code>	831	0.705
<code>pthread_mutex_lock</code>	829	0.672
<code>cma_queue_dequeue</code>	829	0.605
<code>cma_attempt_delivery</code>	696	0.410
Top 10 calls combined		12.64
Total for Pthreads		52.72
Manager		47.28

Table 2: Most frequently called functions

8.3 Influence of System Load

In the following sets of results, the influence of system load on connection setup time is studied. Three sets of measurements were taken. In each set the system load was varied by running a number of CPU-bound processes in competition with the manager. The competing processes each consisted of a shell pipeline³ which required both user space and kernel processing. The competing pipelines were run at the default user priority of 0. The number of interfering pipelines was varied from 0 to 3. For a given system load the performance of the manager was measured by profiling a client which repeatedly set up a connection, exchanged a single message with a remote server, and then tore down the connection. In each experiment, the client set up 100 connections with a random interval between each. In all of the experiments, connections were set up from a DS5000/25 over the ATM network, via 2 Fairisle ATM switch ports, to a second DS5000/25 also running our code. Both workstations were fitted with a simple, cell-based ATM interface [7].

In the first set of measurements, the manager was run at a priority of 0. As expected, the connection setup time degraded as the system load increased. Figure 2 shows an initial sharp increase in connection setup time with 1 competing pipeline, followed by a linear increase with increasing load. In the second set, the priority of the manager was increased to -1. As can be seen from the graph, the connection setup time initially increased rapidly with increasing load, but the rate of increase slowed thereafter. In the third set of experiments the priority of the manager was set at -6, higher than that of the automount daemon. This served to stabilise the performance, in spite of an initial sharp increase with a single competing pipeline.

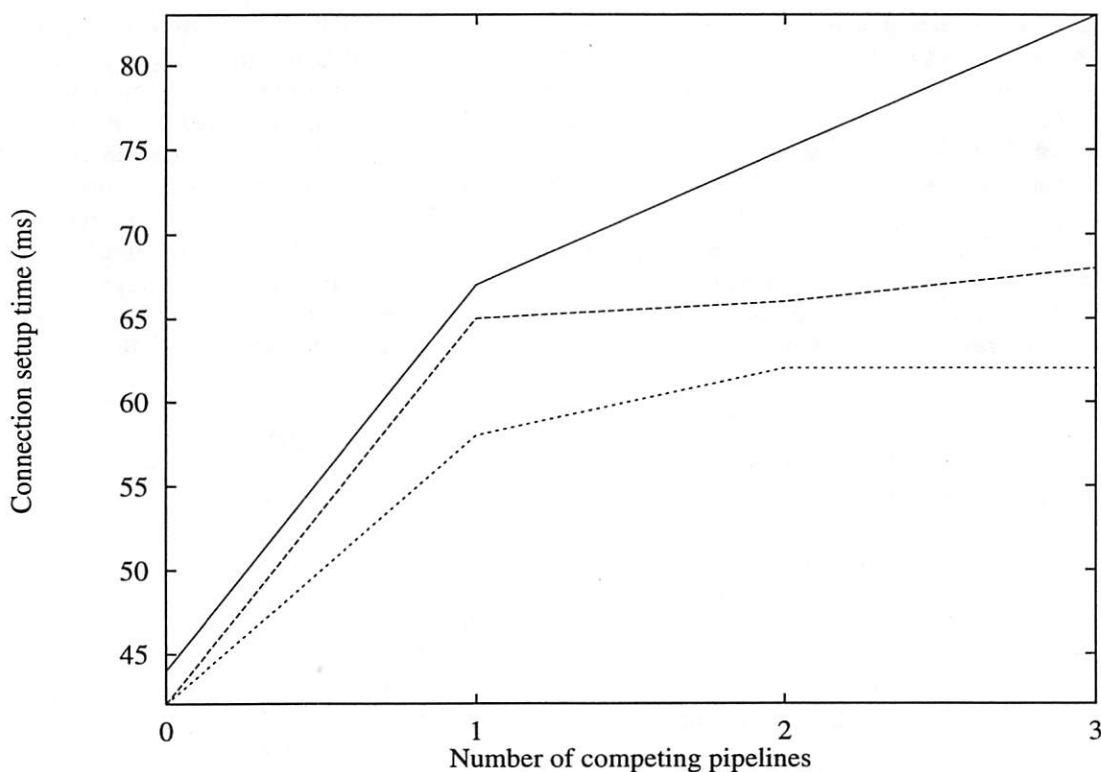


Figure 2: Influence of system load on mean connection setup time

The results suggest that the manager should be run at a high priority, to ensure that an increase in system load does not affect its performance excessively.

³while true; do a='echo \$PATH | grep wanda | sed -e "s/wanda/foo/g"'; echo \$a >/tmp/foo.\$\$; done

8.4 Code Size

An important consequence of our design is that the complexity of the system, in terms of code size, has been significantly reduced. The connection manager compiles to approximately 0.5 MB, almost all of which is due to the Pthreads library. The size of the kernel ATM code has been reduced by about 30 % to a mere 27 KB of text segment including the ATM device driver.

9 Conclusion

This paper has described the design and implementation of an ATM protocol stack within the Unix kernel. A novel approach to the implementation of the control plane has been adopted: the management and control functions are located in a user space daemon. Communication between the kernel networking code and the daemon takes place via a special control socket. The user space daemon is responsible for signalling, socket layer management and control of the ATM protocol code. It is implemented as a multi-threaded process using the Pthreads package.

We conclude that our design is both viable and beneficial. The performance costs are low, and mostly result from a poor Pthreads implementation on Ultrix. The benefits are that the complex and fluid control plane need not be buried in the kernel, and that the resulting code is more portable.

References

- [1] A. Malis, D. Robinson, R. Ullman. Multiprotocol Interconnect on X.25 and ISDN in the Packet Mode. RFC-1356, August 1992.
- [2] ACM. *Fairisle: An ATM Network for the Local Area*, Computer Communications Review, SIGCOMM, September 1991.
- [3] ATM Forum. *ATM User-Network Interface Specification Version 2.1*, 1993.
- [4] M. Burrows. A Prototype Elapsed Time Profiler for Alpha OSF and MIPS Ultrix. Technical report, Digital Equipment Corporation, Systems Research Center, 1993. Yet to appear.
- [5] M. J. Dixon. System Support for Multi-Service Traffic. Technical Report 245, Cambridge University Computer Laboratory, January 1992. Ph.D. dissertation.
- [6] European Fibre Optic Conference. *The Cambridge Backbone Network*, Amsterdam, June 1988.
- [7] L. French, D. Greaves, and D. McAuley. Private ATM Networks and Protocol and Interface for ATM LANs. Technical Report 258, Computer Laboratory, May 1992.
- [8] A. Hopper and R. M. Needham. The Cambridge Fast Ring Networking System. *IEEE Transactions on Computers*, 37(10), October 1988.
- [9] IEEE Computer Society Draft Standard. *Extension for Portable Operating Systems P1003.4a / D6*, 1992.
- [10] Internet Working Group. Draft 5, Classical IP and ARP over ATM, October 14th 1993.
- [11] V. Jacobson. Compressing TCP/IP Headers for Low-Speed Serial Links. RFC-1144, February 1990.
- [12] D. R. McAuley. Protocol Design For High Speed Networks. Technical Report 186, Cambridge University Computer Laboratory, January 1990. Ph.D. dissertation.
- [13] SUN Microsystems. *SunNet X.25 System Administration Manual*, October 1990. Version 7.0 beta.

Richard Black (Richard.Black@cl.cam.ac.uk) obtained a Bachelor's degree in Computer Science from the University of Cambridge in 1990. He is currently in his third year as a Ph.D. student at the University of Cambridge Computer Laboratory. He has worked on the hardware and software of the Fairisle ATM switch, and the implementation of the Wanda micro-kernel.

Simon Crosby (Simon.Crosby@cl.cam.ac.uk) is a Ph.D. student at the University of Cambridge Computer Laboratory, studying control and management of ATM networks. He holds an MSc degree in Computer Science from the University of Stellenbosch, South Africa, and a BSc (Hons) degree in Computer Science and Mathematics from the University of Cape Town, South Africa.

Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages

Masanobu Yuhara
Fujitsu Laboratories Ltd.
1015 Kamikodanaka
Nakahara-ku
Kawasaki 211, Japan

Chris Maeda
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213

Brian N. Bershad
Department of Computer Science
and Engineering FR-35
University of Washington
Seattle, WA 98195

J. Eliot B. Moss
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003

Abstract

This paper describes a new packet filter mechanism that efficiently dispatches incoming network packets to one of multiple endpoints, for example address spaces. Earlier packet filter systems iteratively applied each installed filter against every incoming packet, resulting in high processing overhead whenever multiple filters existed. Our new packet filter provides an associative match function that enables similar but not identical filters to be combined together into a single filter. The filter mechanism, which we call the Mach Packet Filter (MPF), has been implemented for the Mach 3.0 operating system and is being used to support endpoint-based protocol processing, whereby each address space implements its own suite of network protocols. With large numbers of registered endpoints, MPF outperforms the earlier BSD Packet Filter (BPF) by over a factor of four. MPF also allows a filter program to dispatch fragmented packets, which was quite difficult with previous filter mechanisms.

1 Introduction

In this paper, we describe a new packet filter mechanism, which we call MPF (Mach Packet Filter), that efficiently handles many filters that are active at the same time. Our new packet filter also supports context-dependent demultiplexing, which is necessary when receiving multiple packets in a large fragmented message. We have implemented our new packet filter in the context of the Mach 3.0 operating system [Accetta et al. 86]. The new packet filter improves performance by taking advantage of the similarity between filter programs that occurs when performing endpoint-based protocol processing. With 10 TCP/IP sessions, MPF is almost eight times faster than the CMU/Stanford packet filter (CSPF) [Mogul et al. 87], and over four times faster than the BSD packet filter (BPF) [McCanne and Jacobson 93]. MPF's performance advantage grows with the number of sessions.

*This research was sponsored in part by The Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035, by the Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F 19628-91-C-0168, by Fujitsu Laboratories Ltd., the Xerox Corporation, and Digital Equipment Corporation. Bershad was partially supported by a National Science Foundation Presidential Young Investigator Award. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency, Fujitsu Laboratories, Xerox Corporation, Digital Equipment Corporation, the National Science Foundation, or the U.S. Government. Bershad performed this work while at Carnegie Mellon University. Authors' email addresses are {yuhara, bershad, cmaeda, moss}@cs.cmu.edu.

The original packet filters (CSPF and BPF) shared two primary goals: protocol independence and generality. The filters did not depend on any protocol, and future protocols could be accommodated without changing the kernel. MPF shares these two goals, as it is implemented as an extension to the base BPF language. Consequently, a packet filter program built for BPF will work with our system. Although MPF has been implemented for the Mach operating system, it requires no changes to the Mach microkernel interface, and has no Mach-specific aspects. Other BPF implementations could be extended to support MPF programs, and our implementation should port easily to other operating systems that support packet filters.

1.1 Motivation

A packet filter is a small body of code installed by user programs at or close to a network interrupt handler of an operating system kernel. It is intended to carry an incoming packet up to its next logical level of demultiplexing through a user-level process. An operating system kernel implements an interpreter that applies installed filters against incoming network packets in their order of arrival. If the filter accepts the packet, the kernel sends it to its recipient address space. Two packet filters, CSPF and BPF, are common in today's systems. CSPF is based on a stack machine. A CSPF filter program can push data from an input packet, execute ALU functions, branch forward, and accept or reject a packet. BPF is a more recent packet filter mechanism which, instead of being stack-based, is register-based. BPF programs can access two registers (A and X), an input packet (P[]), and a scratch memory (M[]). They execute load, store, ALU, and branch instructions, as well as a return instruction that can specify the size of the packet to be delivered to the target endpoint. BPF admits a somewhat more efficient interpreter than CSPF [McCanne and Jacobson 93].

With a microkernel, where traditional operating system services such as protocol processing are implemented outside the kernel, the original packet filter provided a convenient mechanism to route packets from the kernel to a dedicated protocol server. Scalability was not important because relatively few packet filters would ever be installed on a machine (typically two: one to recognize IP traffic and one to recognize all other traffic). Unfortunately, a single point of primary dispatch for all network traffic resulted in communication overhead for microkernel-based systems substantially larger than for monolithic systems, in which the protocols are implemented in the kernel [Maeda and Bershad 92]. To address this problem, we have decomposed the protocol service architecture so that each application is responsible for its own protocol processing [Maeda and Bershad 93]. That is, every address space contains, for example, a complete TCP/IP stack. Figure 1 illustrates the structural differences between the two different protocol strategies.

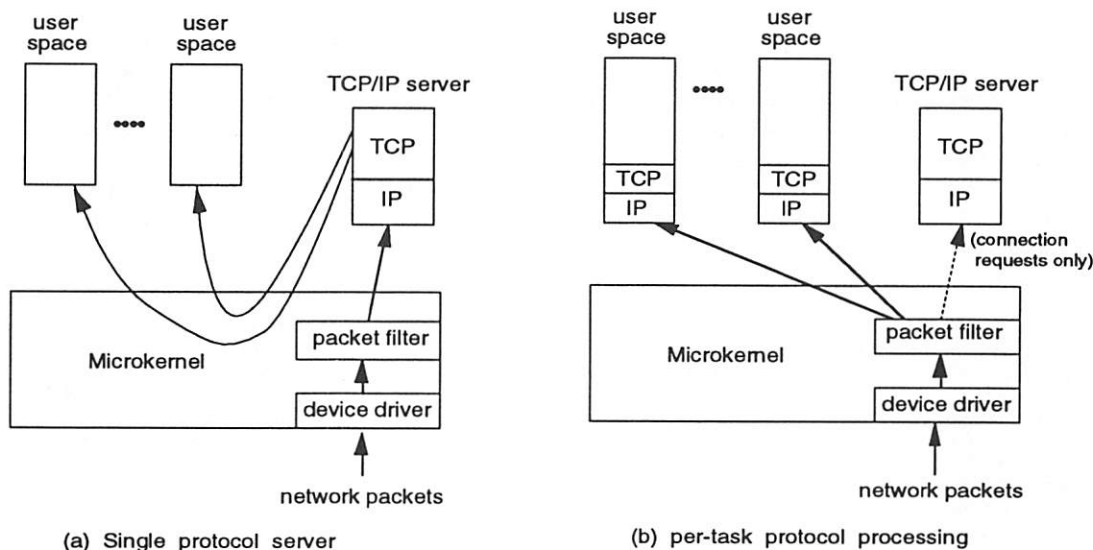


Figure 1: Two ways to structure protocol processing. In the system on the left, all packets are routed through a central server and then on to their eventual destination. In the system on the right, the kernel routes an incoming, but unprocessed network packet directly to the address space for which the packet is ultimately intended, resulting in lower latency and higher throughput. A central server handles operations without critical performance requirements, such as connection establishment.

At its core, our new protocol architecture relies on the kernel's packet filter mechanism to deliver incoming packets

to the appropriate address space. Our application-level protocol processing architecture revealed two serious problems with existing implementations of the packet filter:

1. *The packet filter is not scalable.* The dispatch overhead grew linearly with the number of potential endpoints. For even a workstation-class machine, it is not uncommon to have several hundred protocol endpoints in use at a time, so scalability becomes critical for efficient demultiplexing.
2. *A packet filter is unable to efficiently recognize and dispatch multipacket messages.* Some protocols require information in the previous or future packets to dispatch a packet. For example, the IP protocol splits one large IP packet into several small IP packets when the underlying data link layer cannot accept a large packet [RFC791]. Moreover, the fragmented packets may arrive out of order. The existing packet filters have no mechanisms for efficiently dealing with fragmentation, let alone out of order delivery. Therefore, they cannot dispatch fragmented packets to any of multiple endpoints. Instead, fragments must all be sent to a higher-level intermediary process using the "packet filter of last resort" at the expense of substantially more kernel messages and boundary crossings.

We have solved these two problems by extending the existing BPF instruction set with new instructions that enable the packet filter implementation to support efficiently large numbers of endpoints and fragmented packets.

To deal with the scalability problem, MPF takes advantage of the structural and logical similarity within a protocol, and attempts to dispatch all packets destined for that protocol in a single step. Typically, filter programs for a particular protocol consist of two parts: one that identifies the protocol and one that identifies the session in that protocol. (The code in Appendix A shows an example BPF program for TCP/IP dispatching.) The first part is exactly the same for all sessions within a protocol, while the second part differs only in the constant values that identify the particular session instance. With MPF, the kernel's filter module internally transforms, or *collapses*, filter programs for the same protocol into a single filter program. Figure 2 contrasts MPF with previous packet filter mechanisms, which execute similar code repeatedly for each protocol session.

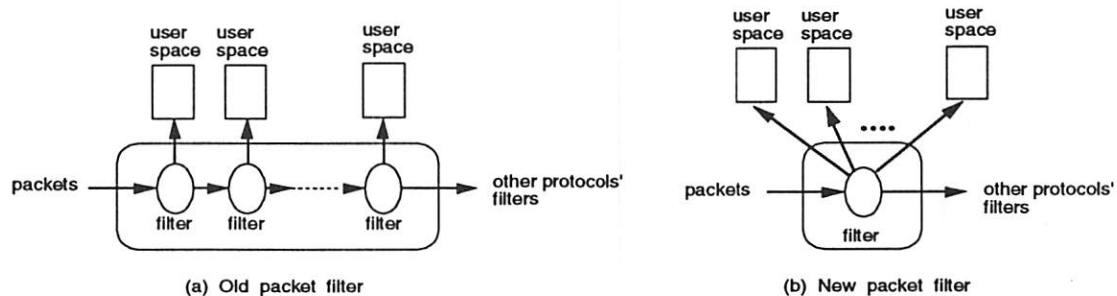


Figure 2: Redundant (BPF and CSPF) vs. one-step filtering (MPF) for incoming packets

To deal with the fragmentation problem, MPF provides per-filter state that persists across the arrival of packets. Filter programs can record dispatch information using an early packet, and later retrieve that recorded information to dispatch a subsequent packet in a multipacket message.

1.2 The rest of this paper

The rest of this paper is organized as follows. In Section 2 we describe the design and implementation of our single-pass packet filter. In Section 3 we present our technique for dispatching fragmented messages. In Section 4 we discuss the performance of our new packet filter mechanism. In Section 5 we present our conclusions.

2 Fast dispatch of incoming packets to multiple endpoints

As mentioned in the previous section, filters generally perform two levels of dispatch. The first level dispatches to a protocol, while the second level to an endpoint within that protocol. The logic for the first level of dispatch is identical for all packets destined to a particular protocol, while that for the second relies on mapping from some number of fields

in the packet header to an actual address space/endpoint. For example, in the MPF implementation, the endpoint is identified by a Mach IPC port [Draves 90], which is explicitly defined when the filter is installed.

We have introduced an associative match instruction (`ret_match_imm`) that allows MPF to exploit the fact that a packet is dispatched first to a protocol and then to a session within that protocol. The `ret_match_imm` instruction, described in Table 1, is a combination of the original packet filter's compare and return instructions.

MPF match sequence	Equivalent BPF match sequence	
<code>ret_match_imm #3, #ALL</code>	<code>ld M[0]</code>	<code>; A = M[0]</code>
<code>key #key0</code>	<code>jeq #key0, k1, fail</code>	<code>; if (A == key0) goto k1; else goto fail;</code>
<code>key #key1</code>	<code>ld M[1]</code>	<code>; A = M[1]</code>
<code>key #key2</code>	<code>jeq #key1, k2, fail</code>	<code>; if (A == key1) goto k2; else goto fail;</code>
	<code>ld M[2]</code>	<code>; A = M[2]</code>
	<code>jeq #key2, ok, fail</code>	<code>; if (A == key2) goto ok; else goto fail;</code>
	<code>ok: ret #ALL</code>	<code>; return the whole packet</code>
	<code>fail: ret #0</code>	<code>; abort this filter</code>

Table 1: The `ret_match_imm` instruction from MPF and its equivalent sequence from BPF. The first argument of `ret_match_imm` indicates the number of data items to be compared. The subsequent `key` pseudo instructions provide immediate data. These immediate values are compared with the values in the scratch memory: `M[0]`, `M[1]`, `M[2]`, respectively. If the corresponding values are equal, then the filter returns with success. The second argument of the `ret_match_imm` instruction specifies the number of bytes of the packet sent to the recipient ("ALL" indicates the entire packet). If any pair of the corresponding values is not equal, the filter terminates with failure, and the packet is not sent to the recipient for this filter.

The MPF implementation uses the `ret_match_imm` instruction to collapse multiple filter programs into one by converting the `ret_match_imm` instruction into a fast associative lookup that precedes the dispatch. When a user task installs a new packet filter, the kernel's filter module scans the program for an associative match instruction. If one is found, the kernel then searches for a previously installed filter program having identical code, but differing only in the immediate values contained in the key pseudo-opcodes following the associative match instruction.

Each suite of similar filters has a hash table in which each entry contains the keys and Mach IPC port of one of the suite's component filters. Upon finding a similar filter, the kernel creates an entry for the new filter in the hash table that corresponds to the similar filter. The kernel then discards the new filter, as it has been effectively collapsed into an existing one. If no match instruction is found, or no similar filter program exists, then the filter is installed with no optimizations applied.

Figure 3 illustrates a sample MPF program that reveals the split-level dispatch and use of the new instruction. The code sequence performs the same function as that in Appendix A. The filter accepts packets sent to a TCP/IP session specified by source IP address (`src_addr`), source TCP port (`src_port`), and destination TCP port (`dst_port`). The combination of these constant parameters is unique for a particular session. Other constant parameters (including the destination IP address, `dst_addr`) are the same for all TCP/IP sessions. The first part (A) of the MPF program checks if the packet uses the TCP/IP protocol. The second part (B) extracts the TCP session information from the packet and puts it into the scratch memory. Parts (A) and (B) are common to all TCP/IP filters. The last part (C) determines if the packet is in fact destined for this particular filter (session).

When a packet arrives and the kernel's filter mechanism processes a collapsed filter, it executes the common part (A and B from the example program) just like a conventional program. If the common part rejects the packet, it is rejected by the whole filter suite, avoiding redundant execution of the filter's common sequence. If the packet is accepted in the common part, the filter module executes the unique part, namely the `ret_match_imm` instruction. Using the values in the scratch memory (`M[0]` .. `M[2]` in the example), the kernel searches the filter's hash table for a match. If the search is successful, then the packet is sent to the corresponding receive port. If the search fails, then the collapsed filter suite rejects the packet (but other filter programs might still apply).

The approach described in this section is powerful and easy to implement, but extremely restrictive. An MPF filter program can have only one common sequence, the filter must begin with the sequence, and no instructions may follow the associative match that marks the end of the sequence. Clearly, more general alternatives could be used. For example, a filter program could have an arbitrary number of common sequences occurring at any location, and any degree of processing outside the common parts could be permitted. Our requirements, though (fast endpoint demultiplexing), combined with the two-level dispatch common to most protocols, suggested a solution that was simple to implement and right most of the time, rather than one that was much more complicated to implement and right about as often.

```

/* Part (A) */
begin                                ; MPF/BPF identifier
ldh    P[OFF_ETHERTYPE]              ; A = ethertype
jeq     #ETHERTYPE_IP, L1, Fail      ; If not IP, fail.
L1:
ld      P[OFF_DST_IP]                ; A = dst IP address
jeq     #dst_addr, L2, Fail          ; If not from dst_addr, fail.
L2:
ldb     P[OFF_PROTO]                 ; A = protocol
jeq     #IPPROTO_TCP, L3, Fail       ; If not TCP, fail.
L3:
ldh     P[OFF_FRAG]                  ; A = Frag_flags|Frag_offset
jset    #!Dont_Frag_Bit, Fail, L4    ; If fragmented, fail.
L4:

/* Part (B) */
ld      P[OFF_SRC_IP]                ; A = src IP address
st      M[0]                        ; M[0] = A

ldxb    4 * (P[OFF_IHL] & 0xf)       ; X = offset to TCP header

ldh     P[x + OFF_SRC_PORT]           ; A = src TCP port
st      M[1]                        ; M[1] = A

ldh     P[x + OFF_DST_PORT]           ; A = dst TCP port
st      M[2]                        ; M[2] = A

/* Part (C) */
ret_match_imm #3, #ALL                ; Compare keys and M[0..2].
key     #src_addr                    ; If matched, accept the
key     #src_port                    ; whole packet. If not,
key     #dst_port                    ; reject the packet.
Fail:
ret     #0

```

Figure 3: An MPF program for a TCP/IP session.

3 Dispatching fragmented messages

Fragmentation occurs when a lower-level protocol layer cannot transfer the entire packet of a higher-level protocol. For example, consider the case of a protocol stack consisting of Ethernet, IP, and UDP. Since UDP messages can be larger than the maximum Ethernet message (4k bytes or larger for NFS packets over UDP, but only about 1500 bytes for Ethernet), they must be sent as a number of IP fragments and reassembled at the destination host. Each IP fragment contains a common message id that is unique to the UDP message, offset and length information that describe which part of the UDP message the fragment contains, and finally the data. Only the first fragment contains the UDP header, which includes the UDP source and destination port numbers. The message id, offset, and length are used by IP to reassemble the incoming UDP message.

Demultiplexing incoming fragments is difficult for several reasons: only the first fragment contains the transport protocol header which provides the information needed to determine the target endpoint, fragments may arrive out of order, and some fragments may not arrive at all. We wanted to support simple and efficient demultiplexing of fragments using the packet filter. We were not concerned with performing actual reassembly at the packet filter layer, as we expected that service to be provided by user-level code.

To deal with fragmentation, we introduced a per-filter memory that allows packet filters to link dispatch information present in only the first fragment of a message (for example, the UDP port numbers), and information present in all fragments (for example, the unique message id).¹ Filter programs can record the higher level session dispatch information and associate it with a lower level message id, allowing them to dispatch fragments to the correct endpoint. This association persists for a finite time, after which it is automatically removed.

Because fragments don't always arrive in order, we also allow a filter to postpone processing of a fragment when the first fragment of a message is not the first to arrive. Since no dispatch information is available for the earliest arriving fragments, these fragments are postponed, and processed only after other packets have arrived. Hopefully, the

¹We assume that the fragments of a message between any pair of source and destination addresses share a message id that is unique to the message.

dispatch information will have become available. Postponed packets are dropped if it appears the dispatch information will not become available, or if space to record the postponed packets is exhausted.

3.1 Details of the approach

We have added four new instructions to the packet filter instruction set to handle fragmentation: `register_data`, `ret_match_data`, `jmp_match_imm`, and `postpone`. These instructions are described in Table 2, and demonstrated in Appendix B which shows the flow of a UDP/IP filter program that handles fragmented packets.

Instruction	Description
<code>jmp_match_imm #N, Lt, Lf</code>	This instruction is used to identify the first of a fragmented packet. The instruction is similar to the <code>ret_match_imm</code> instruction in that the N immediate data values following the instruction are compared with M[0] .. M[N-1] of the scratch memory. This instruction conditionally jumps forward depending on the result of the comparison. If the data match, control transfers to Lt, otherwise control transfers to Lf.
<code>register_data #N, #T</code>	This instruction is used to bind an endpoint to a message id. The scratch memory values M[0]...M[N-1], intended to be the message id, recorded in the filter's static memory. These values are removed after T milliseconds. T should be longer than the expected time to receive all fragments in a message. This instruction may only execute following a successful <code>jmp_match_imm</code> , which creates the higher association between a particular named endpoint and the first fragment destined for that endpoint.
<code>ret_match_data #N, #R</code>	This instruction is used to return a fragment in a large message to the appropriate endpoint. The instruction compares M[0]..M[N-1] of the scratch memory, intended to contain the message id of the current packet, with the static memory of this filter. If the values are the same, R bytes of the packet are sent to the recipient of this filter. If not (or if the static memory values do not exist), execution continues with the next instruction.
<code>postpone #T</code>	This instruction postpones processing of the current packet, deferring it to some later time. If a postponed packet is chosen for processing, it may be postponed again. The packet is discarded after T milliseconds from its original arrival, although it may be discarded earlier because of storage limitations.

Table 2: New instructions to support handling of fragmented packets. All but the `jmp_match_imm` instruction are expected in the common part of a filter, enabling the collapse of filters that dispatch fragments.

The `jmp_match_imm` instruction is a branching version of the `ret_match_imm` instruction described in the previous section. If the match fails, the program branches to the false-case label. If the match succeeds, the program branches to the true-case label. As a side effect of matching, the receive port associated with the key data following the `jmp_match_imm` instruction becomes associated with the currently running packet filter. If the filter then executes a normal return instruction, the associated receive port is recalled and used as the recipient. In this way, we avoid having to explicitly manipulate kernel descriptors (really, IPC ports) within the packet filter, yet are able to collapse filters that handle fragmentation.

The `register_data` and `ret_match_data` instructions store and retrieve the fragmentation information. When a packet filter executes the `register_data` instruction, the contents of its scratch memory are used as keys associated with its receive port in a second (filter-specific) hash table. The `ret_match_data` instruction uses this hash table to provide fast lookup on the fragment information. Each entry in the second hash table has its own expiration time specified by the filter program.

The `postpone` instruction addresses the situation where a later fragment arrives before the first fragment. We assume that out-of-order arrival is rare, and use a simple postponement mechanism. A postponed packet is placed on a pending packet queue. Pending packets are reprocessed immediately after each new packet is filtered. Of course, the filter program may postpone the packet again. However, the packet's expiration time is set when it is first postponed, and the packet will be dropped after that time, or if the number of postponed packets becomes too large.

The fragmentation support described in this section imposes no overhead for filtering non-fragmented packets. Note that the kernel does not do reassembly; that work occurs in the endpoint address space itself. The kernel merely assists in routing the fragments to the appropriate reassembly routine.

4 Performance

In this section we discuss the performance of our new packet filter mechanism by comparing it to BPF and CSPF. We answer four questions about performance:

- how does filter processing overhead grow as a function of the number of installed protocol endpoints?
- what is the round-trip latency for messages received through the packet filter as a function of the number of installed protocol endpoints?
- what is the overhead to install a new filter?
- how long does it take to detect and handle a fragmented message?

We conducted our measurements on a DECstation 5000/200 (25 MHz MIPS R3000, 64 KB instruction cache, 64 KB data cache, Lance AMD7990 Ethernet controller) running Mach 3.0 (MK82) and CMU's Unix single-server server (UX41). All timing measurements were taken using a memory-mapped 25 MHz free-running counter.

4.1 Packet filter latency

The most important performance metric for our new packet filter mechanism is its scalability, which we measure in terms of filtering latency as a function of the number of active sessions. By latency, we mean the time required for the packet filter module to determine which filter accepts an incoming packet. This time is constant for all packet sizes, since the filters examine only the packet headers. Latency does not include the time spent in the kernel's interrupt handler, message transmission time, or the time to dispatch a matched packet to an address space; these times are constant for all the packet filter mechanisms. To put the numbers in this section in context, Table 3 shows the time required for the Ethernet device driver to service an incoming packet, and for the kernel to deliver it to the destination address space. These operations occur before and after the packet filter runs, respectively. Packet filtering (demultiplexing) should take much less time than these operations, which are dominated by data movement.

Operation	Packet Size	
	64	1514
Time to read an incoming packet from the Ethernet device.	0.1	0.5
Time to move an incoming packet from the kernel to the destination address space.	0.1	0.2

Table 3: *The time (in milliseconds) required for the device driver to service incoming packets and for the kernel to deliver an incoming packet to its destination address space. These times were measured on a DECstation 5000/200 running Mach 3.0.*

Our latency tests reflect just that overhead required to interpret and initiate a packet dispatch to the receiver's address space. To isolate cache effects, we ran the controlled benchmark with cold and warm caches. Worst-case (cold-cache) measurements were obtained by flushing the caches before applying a filter. Best-case (warm-cache) measurements were obtained by running the benchmark without flushing the caches. Actual performance will vary between these extremes, with the operating point depending on the frequency of packet arrival, and the nature of other system activity.

We varied the number of TCP/IP filters installed in the filter module (no other protocols were registered), and measured the latency for the three packet filter implementations. For MPF, all filters could be collapsed into a single filter. Figure 4 shows the results. The latency of both BPF and CSPF grows linearly as the number of sessions increases because both filter mechanisms must run a filter program for each session. The latency for MPF, on the other hand, is insensitive to the number of sessions since only one filter program is executed to demultiplex packets for all sessions. With only ten TCP sessions, MPF (0.035 ms) shows performance that is 7.8 times better than CSPF (0.273 ms) and 4.3 times better than BPF (0.152 ms) in the warm cache case. At 100 sessions, the total demultiplexing time for either BPF or CSPF (add the times in Table 3 to the filtering time) is on the order of the round trip time for many finely tuned network protocol implementations [Schroeder and Burrows 90].

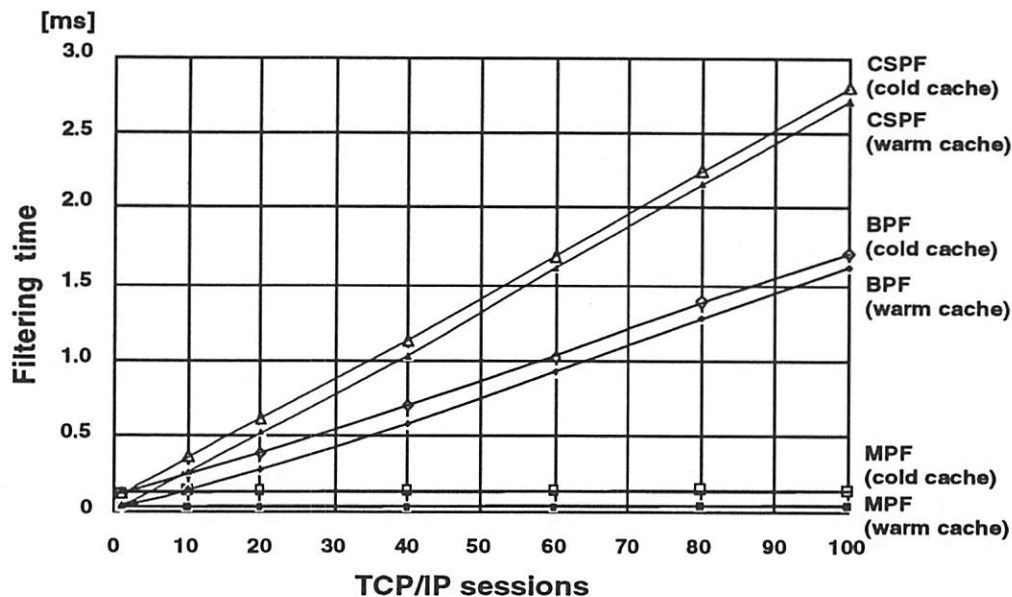


Figure 4: Filtering time of the three filter mechanisms.

4.2 Round-trip times

To observe the impact on end-to-end application performance, we measured the round-trip time of UDP/IP packets when the protocol service is implemented in the application's address space [Maeda and Bershad 93]. We ran two DECstation 5000/200's in single user mode, connected via private Ethernet. One of the hosts sent UDP packets with one data byte to the other, which replied with the same message. Each host installed the same number of filters. Packets were sent repeatedly, therefore this experiment roughly corresponds to the warm-cache case in Figure 4.

The Mach 3.0 kernel dynamically reorders packet filters so that more active filters are processed before less active ones. We disabled filter reordering so that the filter for the active session was always processed last. This gives us a worst-case estimate of round-trip latency for CSPF and BPF, rather than the best-case that would be given by reordering. In practice, the actual latency will lie somewhere between the best and the worst case, and will be determined by traffic patterns. In contrast, MPF latency is independent of the traffic patterns, and has the same best-case and worst-case latency.

Figure 5 shows the round-trip time of a UDP packet as observed by the application program. The time includes protocol stack processing, interprocess communication within a host, and physical network transfer, as well as packet filtering. MPF is 12% faster than BPF with only 10 sessions, and is over four times faster than BPF with 100 sessions.

In contrast to the filtering latency itself (Figure 4), the round-trip time for UDP using MPF grows slightly as more filters are added. This is due to the combination of our experimental methodology, and our implementation of the Internet protocols. We create 100 filters by creating 100 Unix processes, each with its own UDP endpoint, and each with its own version of an Internet protocol library (UDP/IP and TCP/IP). The library includes several periodic threads that ran during the experiment. These threads create an artificial load on the machine, increasing the wakeup latency of the thread that performs the actual protocol processing, thereby increasing the round-trip time.² CSPF and BPF are similarly affected but by no more than the slope indicated with MPF.

4.3 Filter installation overhead

As described in Section 2, the MPF module must analyze a filter program during installation. Consequently, the overhead of installing a filter under MPF might be larger than that for BPF or CSPF, which do not collapse similar filters. Two aspects of installation are important: the time to install a new filter (session) that uses a protocol for which a previous filter has already been installed and the time to install a new filter that uses a protocol for which no previous filter has been installed.

²This background activity is a bug in the library implementation, which is being fixed for the next release.

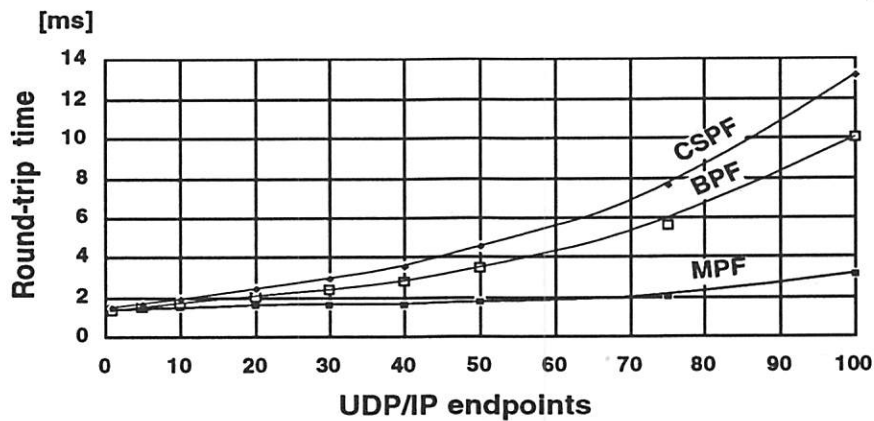


Figure 5: UDP/IP round-trip time as observed by an application program as a function of the number of installed endpoints.

We measured filter installation overhead by installing, one-by-one, new filters that were all sessions of a single protocol (first case, Figure 6), or that were all sessions of different protocols (second case, Figure 7). Both cases show the installation time to be comparable. In the first case (many filters, same protocol), when the number of installed programs is small, BPF and CSPF filter installation time is about twice as fast as MPF, because MPF must compare the new filter program against each installed program. As the number of filters increases, the performance difference diminishes; with 75 sessions, the installation time is about the same for the three filter mechanisms. All filter mechanisms must search previously installed filters to determine whether a new filter is being installed, or an existing one replaced. The use of a more efficient internal representation (hash table vs. linked list) in MPF allows the installation time to grow more slowly.

In the second case (many filters, different protocols) the installation time for MPF grows linearly with the number of filters while the installation time for BPF and CSPF remains constant. The increase is because MPF tries to match each new filter with each existing filter, fails, and then prepares a separate hash table for the new filter. In practice, though, we expect that the number of protocols (and so the number of uncollapsed filter programs) to be small.

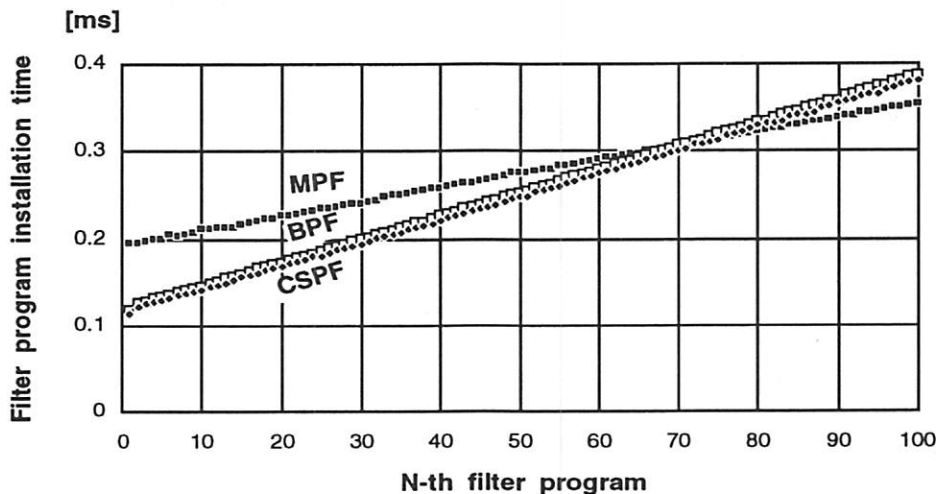


Figure 6: Filter program installation time. All filters are for the same protocol.

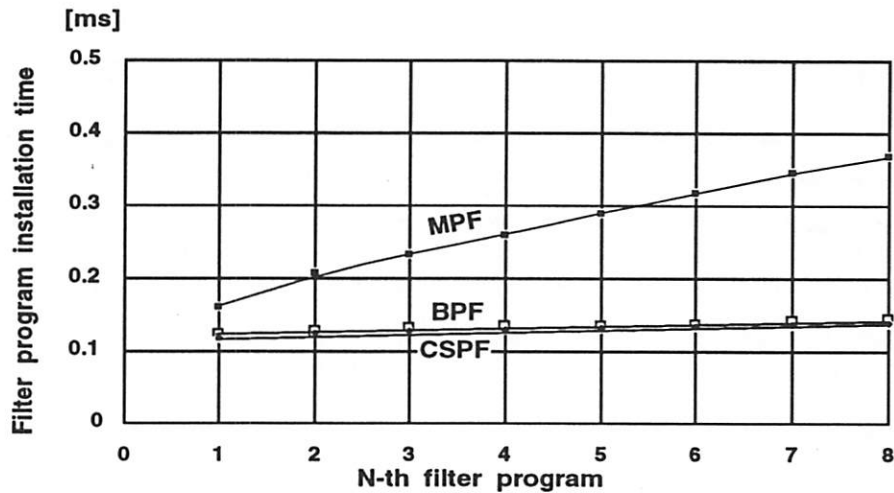


Figure 7: Filter program installation time. Each filter is for a different protocol.

4.4 Fragmentation overhead

Since neither CSPF nor BPF supports fragmentation, protocol implementations that use them must install “default” packet filters to route message fragments to a dedicated server, where they are reassembled and forwarded to their final destinations. In contrast, MPF provides support to route fragments directly to their final destinations, eliminating the intermediate processing in the server.

Table 4 shows filtering latency for fragmented UDP/IP packets using the MPF program shown in Appendix B. When message fragments arrive in order, filtering time for the second and later fragments is no different from filtering time of a non-fragmented packet. For the first fragment, though, MPF needs more time to register the association between the message id and the destination endpoint. When message fragments arrive out of order, the fragments are processed and postponed until they can be dispatched to their final destination. When the first fragment arrives, the total time to process it and any pending fragments is the sum of the filtering time required for those packets when they arrive in order.

Packet	Incoming fragment	Fragment processing	[ms]
Non-fragmented	—		0.04
Fragmented (in-order)	1st	Register, dispatch	0.07
	2nd or later	Dispatch	0.04
Fragmented (out-of-order)	2nd	Postpone	0.04
	1st	Register, dispatch (for the 1st), retry, dispatch (for the 2nd)	0.11

Table 4: The filtering time of non-fragmented and fragmented UDP/IP packets. Two cases for fragmented packets are shown: a case for in-order arrival (the first fragment arrives, then the second fragment), and a case for out-of-order arrival (the second fragment, and then the first fragment).

As mentioned, the main advantage of MPF’s support for fragmentation is that fragmented messages do not have to be routed through an intermediate server. Using the UDP round-trip time program described earlier and 2048 byte packets (two fragments on an Ethernet), we measured a round-trip time of 11.1 ms. for an MPF-based UDP, 11.8 ms. for one based on BPF, and 11.9 ms. for one based on CSPF. CSPF and BPF have similar performance because it is dominated by the time required to move the data from the kernel to the server for reassembly, and from the server to the application. The 7% improvement for MPF comes from avoiding the indirection through a central server.

5 Conclusions

MPF is a new packet filter mechanism that can efficiently dispatch small and large packets even in the presence of many sessions, making it suitable for per-task protocol processing. We have introduced a new match instruction that the packet filter mechanism can use as a hint to collapse similar packet filters into one. Collapsing removes repetitive execution of the same code. MPF also supports new instructions to dispatch fragmented packets. Our implementation of MPF is 7.8 times faster for TCP/IP packet filtering than CSPF, and 4.3 times faster than BPF with only ten registered sessions. The source code for MPF can be obtained through anonymous ftp as part of CMU's Mach 3.0 distribution at *mach.cs.cmu.edu*.

References

- [Accetta et al. 86] Accetta, M.J., Baron, R. V., Bolosky, W., Golub, D. B., Rashid, R. F., Tevanian, Jr., A., and Young, M.W., "Mach: A New Kernel Foundation for UNIX Development", Proceedings of the Summer 1986 USENIX Conference, pp.93-113, July 1986.
- [Draves 90] Draves, R. "A Revised IPC Interface", Proceedings of the First Mach Workshop, pp. 101-121, October 1990.
- [Maeda and Bershad 92] Maeda, C., and Bershad, B.N., "Network Performance for Microkernels", Proceedings of the Third Workshop on Workstation Operating Systems, April 1992.
- [Maeda and Bershad 93] Maeda, C., and Bershad, B.N., "Protocol Service Decomposition for High-Performance Networking", The Proceedings of the 14th ACM Symposium on Operating Systems Principles, December 1993.
- [McCanne and Jacobson 93] McCanne, S., Jacobson, V., "The BSD Packet Filter: A New Architecture for User-level Packet Capture", Proceedings of the Winter 1993 USENIX Conference, pp.259-269, January 1993.
- [Mogul et al. 87] Mogul, J., Rashid, R., and Accetta, M., "The Packet Filter: An Efficient Mechanism for User-level Network Code", Proceedings of the 11th ACM Symposium on Operating Systems Principles, pp.39-51, 1987.
- [RFC791] Postel, J. B., "Internet Protocol", Request For Comments 791, September 1981.
- [Schroeder and Burrows 90] Schroeder, M. and Burrows, M., "Performance of Firefly RPC", ACM Transactions on Computer Systems (8)1, pp.1-17, February 1990.

Appendix

A BPF example program

```
/*
 * P[i]: packet data at byte offset i.
 * M[i]: i-th word of the scratch memory.
 * Word = 4 Bytes, Half Word = 2 Bytes, Byte = 1 Byte.
 *
 * dst_addr: IP address of this host
 *           (destination IP address of this session)
 *
 * src_addr: source IP address of this session
 * src_port: source TCP port number of this session
 * dst_port: destination TCP port number of this session
 */

begin                                ; BPF identifier
ldh  P[OFF_ETHERTYPE]                ; A = ethertype
jeq  #ETHERTYPE_IP, L1, Fail        ; If not IP, fail.
L1:  ld  P[OFF_DST_IP]                ; A = dst IP address
jeq  #dst_addr, L2, Fail            ; If not from dst_addr, fail.
L2:  ld  P[OFF_SRC_IP]                ; A = src IP address
jeq  #src_addr, L3, Fail            ; If not from src_addr, fail.
L3:  ldb  P[OFF_PROTO]                ; A = protocol
jeq  #IPPROTO_TCP, L4, Fail        ; If not TCP, fail.
L4:  ldh  P[OFF_FRAG]                ; A = Flags|Frag_offset
jset #!Dont_Frag_Bit, Fail, L5     ; If fragmented, fail.
L5:  ldxb 4 * (P[OFF_IHL] & 0xf)      ; X = offset to TCP header
ldh  P[x + OFF_SRC_PORT]             ; A = src TCP port
jeq  #src_port, L6, Fail            ; If not from src_port, fail.
L6:  ldh  P[x + OFF_DST_PORT]         ; A = dst TCP port
jeq  #dst_port, Suc, Fail           ; If not to dst_port, fail.
Suc: ret #ALL                        ; Accept the whole packet.
Fail: ret #0                        ; Reject the packet.
```

B An example filter program that processes fragmented packets

```
/*
 * P[<x>]: Data <x> in the packet.
 * M[i]: i-th word of the scratch memory.
 */
begin                                ; BPF identifier
ldh  P[OFF_ETHERTYPE]                ; A = ethertype
jeq  #ETHERTYPE_IP, L1, Fail        ; If not IP, fail.
L1:  ldb  P[OFF_PROTO]                ; A = protocol
jeq  #IPPROTO_UDP, L2, Fail        ; If not UDP, fail.
L2:  ldh  P[OFF_FRAG]                ; A = Flags|Frag_offset
jset #0x1fff, Frag2, L3            ; non-zero Frag_offset?
L3:  /*
   * Frag_offset is zero:
   * Packet is either a 1st frag
   * or an unfragmented datagram.
   * Now we can check the UDP header.
   */
ld  P[OFF_DST_IP]                   ; A = dst IP address
st  M[0]                           ; first word of key
```

```

ldxb    4 * (P[OFF_IHL] & 0xf) ; X = offset to UDP header
ldh     P[x + OFF_DST_PORT]    ; A = dst UDP port
st      M[1]                  ; second word of key

/*
 * Now that the session key is in
 * the scratch memory, we check to
 * see if packet is a fragment that
 * needs to associate the message ID
 * with an endpoint or an unfragmented
 * message that can simply be delivered.
 */
ldh     P[OFF_FRAG]            ; A = Flags|Frag_offset
jset    #0x2000, Frag1, NotFrag ; is More_fragment set?
Frag1:
/*
 * First fragment:
 * Match on UDP session key then
 * register the Message ID.
 */
jmp_match_imm #2, RegData, Fail
key     #dst_addr
key     #dst_port

RegData:
/*
 * Register the message ID in
 * the per-filter hash table.
 */
ldh     P[OFF_IP_ID]           ; A = IP Message ID
st      M[0]
ld      P[OFF_SRC_IP]          ; A = IP source address
st      M[1]
register_data #2, #timeout      ; Associate key with this filter
ret     #ALL                   ; Deliver first fragment

NotFrag:
/*
 * Normal unfragmented datagram.
 */
ret_match_imm #2, #ALL
key     #dst_addr
key     #dst_port

Frag2:
/*
 * Other fragment:
 * Build a key and look in
 * the per filter hash table.
 */
ldh     P[OFF_IP_ID]           ; A = IP Message ID
st      M[0]
ld      P[OFF_SRC_IP]          ; A = IP source address
st      M[1]
ret_match_data #2, #ALL        ; return ALL if match
postpone #timeout              ; postpone if no match

Fail:
ret     #0

```


Latency Analysis of TCP on an ATM Network

Alec Wolman, Geoff Voelker, and Chandramohan A. Thekkath
Department of Computer Science and Engineering
University of Washington

Abstract

In this paper we characterize the latency of the BSD 4.4 alpha implementation of TCP on an ATM network. Latency reduction is a difficult task, and careful analysis is the first step towards reduction. We investigate the impact of both the network controller and the protocol implementation on latency. We find that a low latency network controller has a significant impact on the overall latency of TCP. We also characterize the impact on latency of some widely discussed improvements to TCP, such as header prediction and the combination of the checksum calculation with data copying.

1 Introduction

In this paper we investigate the latency characteristics of the TCP transport protocol on an Asynchronous Transfer Mode (ATM) network[6]. The characteristics of LAN technologies have changed a great deal in the last few years. With faster network hardware, the disparity between software and hardware costs is even greater. This increases the importance of efficient protocol implementations and efficient operating system interfaces. The following factors in network communication make measuring TCP performance, especially latency, interesting:

- The existence of a high quality TCP software implementation: the BSD 4.4 alpha TCP code.
- The availability of low latency network interfaces: e.g., the FORE TCA-100 ATM interface[6].
- The wide use of applications and subsystems (like RPC) that can benefit from reduced latency.

Prior studies have concentrated on characterizing and optimizing the throughput of TCP on substantially different hardware or networks than the ones we describe here (e.g., [3, 8]). In addition to focusing on an ATM network, we investigate how optimizations previously suggested for improving throughput affect latency. We believe that studying the latency characteristics of TCP on ATM networks is particularly interesting for two reasons. First, ATM is an emerging communication standard that is likely to be widely deployed. Second, our study allows us to answer the following questions: Can we provide evidence that TCP is a viable option for a transport layer for RPC? How have the changes in technology affected the results of earlier studies (e.g., [4])? Is latency dominated by the cost of operating system services, such as buffer management? If so, can the use of such services be reduced enough to make latency acceptable for applications that require low latency?

1.1 System Overview

All of our experiments were run on a pair of DECstation 5000/200 workstations, which use a MIPS R3000 processor running at 25 MHz. Each DECstation was equipped with a FORE TCA-100 ATM network interface on the TurboChannel I/O bus. The ATM network interface uses a memory mapped receive FIFO that stores up to 292 53-byte ATM cells, and a similar transmit FIFO that stores up to 36 cells. The transmit engine starts reading from the

This work was supported in part by the National Science Foundation under Grants No. CCR-8907666, CDA-9123308, and CCR-9200832, by the Washington Technology Center, Apple Computer, Boeing Computer Services, Digital Equipment Corporation, and the Hewlett-Packard Corporation. Chandramohan A. Thekkath was also supported by an Intel Foundation Graduate Fellowship.

transmit FIFO as soon as there is one complete cell in the FIFO. The ATM driver and adapter implement the Class 3/4 ATM Adaptation Layer (AAL), which is responsible for all segmentation and reassembly of datagrams and the detection of transmission errors and dropped cells.

We used the ULTRIX 4.2A kernel as a foundation and replaced its TCP implementation with the BSD 4.4 alpha TCP implementation. The BSD TCP implementation has lower latency than the ULTRIX implementation, in part because of its use of one fewer mbuf for small packets and its use of a protocol control block cache¹. Since ULTRIX 4.2A is a BSD derivative, substituting the new BSD TCP implementation was relatively straightforward because the two implementations have nearly identical interfaces to the rest of the protocol stack.

1.2 Measurement Techniques

Many of our experiments measured the round-trip latency of two user-level processes roughly simulating client-server communication. Unless stated otherwise, the processes ran on otherwise idle machines and communicated over a switchless private ATM network. The client connected to the server using TCP, started a timer, and then repeatedly executed the following steps: it sent *size* bytes to the server, and then waited to receive *size* bytes from the server. It then stopped the timer and recorded its value. For all the round-trip measurements in this paper, we ran 40000 iterations for at least 3 repetitions and took the average to get the final result.

Our measurements used a range of packet sizes. Based upon previous studies of RPC and TCP traffic behavior, we chose a variety of packet lengths sized 500 bytes and smaller [1, 8]. We also measured packets of 1400 bytes (the Ethernet MTU minus protocol headers), 4000 bytes (fits on a single memory page including protocol headers), and 8000 bytes (fits on two memory pages including protocol headers, also close to our ATM MTU of 9K).

Latency measurements typically involve estimates of small code paths that take on the order of microseconds. To measure at this level of granularity, we used a real time clock on a TurboChannel card with a 40ns period². One advantage of using this clock is that we avoid instruction counting as a technique for estimating the execution time of small code sections. One disadvantage of this approach, however, is that our measurements include cache effects.

The clock is initialized at boot time, and user-level processes gain access to it by issuing a system call that maps the clock address into the process's address space. Reading the clock is then just a matter of dereferencing a pointer. Code inside the kernel can read the clock in a similar manner. We also added system calls to extract timings from the kernel to measure events that started in user space and ended in the kernel, or vice-versa.

1.3 Paper Outline

The rest of this paper is organized as follows. Section 2 summarizes our measurements of TCP latency on the baseline system. Sections 3 and 4 study the effect of several modifications motivated by the results in Section 2. These modifications are not new and have been suggested by others to improve throughput[4, 9]. However, our focus here is on the effect of these modifications on latency.

2 Measurement of the Baseline System

The baseline system that we measured is the BSD 4.4 alpha TCP release operating on an ATM network. From our measurements, we investigate: (1) the contribution to latency of the network driver and adapter; (2) the cost of TCP protocol processing; and (3) the overhead of protocol-independent operating system mechanisms.

2.1 Effect of the Network on Latency

To demonstrate the effects of the network driver, adapter, and physical link on latency, we compared the round-trip times of the BSD 4.4 TCP implementation communicating over the ATM network with the same TCP implementation communicating over Ethernet. The results are listed in Table 1. For the small transfer sizes, the network has a large effect on overall latency (e.g., a 919 μ s difference in the 4 byte case). For large transfer sizes, much of the effect can be attributed to the lower bandwidth of the Ethernet driver, adapter, and physical link.

¹ A comparison of the two implementations can be found in University of Wash. CSE Dept. Tech. Report #93-03-02.

² The TurboChannel card is the AN-1 controller from DEC SRC[16]. Note that we did not employ the AN-1 network in this study, only the clock on its controller.

Size (bytes)	Round Trip Times (μ s)		Percentage Decrease (%)
	Ethernet	ATM	
4	1940	1021	47
20	2337	1039	55
80	2590	1289	50
200	2804	1520	45
500	4101	2140	47
1400	6554	2976	54
4000	13168	5891	55
8000	22141	10636	52

Table 1: Comparison of ATM versus Ethernet latencies.

2.2 Detailed Measurements of Latency

To obtain detailed latency measurements, we instrumented the transmit and receive sides separately. We used the same benchmark program described above to measure both sides. The results for the transmit side are shown in Table 2, and the results for the receive side are shown in Table 3.

Layer		Latency (μ s)							
		Packet Size (bytes)							
		4	20	80	200	500	1400	4000	8000
User		45	45	48	67	121	99	174	400
TCP	checksum	10	12	23	42	90	209	576	1149
	mcop	5.1	5.7	26	41	80	29	30	41
	segment	62	65	63	65	71	63	65	72
	Total	77	81	112	148	241	301	671	1262
IP		35	34	35	35	36	36	38	36
ATM		23	24	39	47	71	96	215	498
Total		180	184	234	297	469	532	1098	2196

Table 2: Breakdown of BSD 4.4 alpha Transmit Side Latency

In characterizing the latency of transmitting data using TCP, we divided the transmit operation into four time spans. The first span, **User**, measures the time from the *write* system call to the beginning of the TCP protocol implementation. This span of time includes copying data from user space into kernel mbufs at the socket layer.

The second span, **TCP**, measures the time spent doing the TCP protocol output processing. It consists of three components, **checksum**, **mcop**, and **segment**. **Checksum** is the time spent calculating the TCP checksum over the data and header. **Mcopy** is the time spent copying data from the socket mbufs into driver mbufs. **Segment** is the remaining TCP protocol processing time.

The third time span, **IP**, measures the time spent in IP output processing, and the last span, **ATM**, measures the time spent in the ATM network driver. To obtain an accurate measurement of latency for the last span, we only measure up to when the ATM adapter is signaled to send the last byte of data. We do not include the time of any operations after that because these operations are effectively overlapped with network transmission, which is separately accounted for.

The rows in Table 3 have similar meanings. The **User** time span refers to the time from when the data leaves the TCP layer until the time the user process runs again (except for the scheduling time, described below). **TCP** is the time spent doing the TCP input processing, and has a similar breakdown as on the transmit side. Note, however, that the TCP input processing does not have a **mcop** row because the extra copy operation is only used on the transmit side to support retransmissions. **IP** is the time spent doing IP input processing, and **ATM** is the time spent receiving and reassembling incoming ATM cells.

Layer		Latency (μ s)							
		Packet Size (bytes)							
		4	20	80	200	500	1400	4000	8000
ATM		46	46	70	99	164	363	920	1783
IPQ		22	22	22	22	23	45	46	50
IP		40	40	62	62	62	53	54	43
TCP	checksum	10	12	23	40	82	211	578	1172
	segment	135	135	138	141	158	142	143	59
	Total	145	147	161	181	240	353	721	1231
Wakeup		46	47	47	50	49	51	58	67
User		64	65	89	81	102	124	199	468
Total		363	367	451	495	640	989	1998	3642

Table 3: Breakdown of BSD 4.4 alpha Receive Side Latency.

We also introduced two more time spans on the input side. The first, **IPQ**, measures the IP queue scheduling time, i.e., the time from when the ATM driver places received data on the IP queue and signals a software interrupt until the time the data is removed from the IP queue. The second, **Wakeup**, is the user process scheduling time, i.e., the time from when the user process is placed on the run queue until the time it runs.

The nonlinear response of the ATM adapter, as observed in the **ATM** rows of the receive data, is due to overlap between sending the data and receive processing. When the sending ATM adapter is sending a large number of cells, the receiving ATM adapter can process the first cells while the sending adapter is still sending the later cells. We only measure the portion of the receive processing that actually contributes to the overall latency. This is the time from the arrival of the last group of ATM cells comprising the last TCP segment of a data transfer to the time when the *read* system call returns to the user-level process. We use the arrival of the last group of ATM cells comprising the last TCP segment to initiate our timings because we know at that point that the sending adapter has finished sending all of the data for that transmission.

The following subsections present an analysis of the data in these tables.

2.2.1 Mbuf Manipulation

One to eight mbufs are used for transfers of less than 1 KB. Beyond this size, cluster mbufs are used. Cluster mbufs are used for large transfers because they hold 4 KB of data, the size of a memory page, whereas normal mbufs hold only 108 bytes of data. The measured time to allocate and free an mbuf (independent of type) is just over 7 μ s, making the mbuf manipulation a small cost relative to the overall cost of sending or receiving data.

The nonlinear response between the 500 and 1400 byte transfer sizes of the **User** and **mcopy** rows of Table 2 is due to a switch in the use of mbuf types in the ULTRIX 4.2A socket layer. Once the data transfer size grows above 1 KB, ULTRIX uses cluster mbufs to store user data.

In the **User** row, the copy from the user buffer to the mbuf takes less time because user data does not have to be fragmented into multiple mbufs.

In the **mcopy** row, using cluster mbufs reduces latency because the mbuf-to-mbuf copy semantics of cluster mbufs differs from normal mbufs. When normal mbufs are copied, the data is actually copied into separately allocated mbufs. However, cluster mbufs use reference counts for copying; no storage is allocated or data copied. Since TCP makes a copy of the mbufs passed from the socket layer on the transmit path, the copy for transfers larger than 1 KB takes less time than for smaller transfers.

We note, however, that these effects are artifacts of a particular buffer management implementation choice rather than inherent protocol behavior.

2.2.2 Checksum

The checksum does not scale linearly with the small transfer sizes because the checksum is done over the data and the TCP/IP header (20 bytes for TCP header + 20 bytes for IP overlay + length of TCP options). Also, as transfer sizes grow, the checksum calculation begins to dominate the cost of protocol processing. In a later section, we discuss optimizing the checksum for better latency.

2.2.3 Data Copies

The times in three rows of the tables (**User**, **mbcopy**, and **ATM**) include the cost of a data copy: the **User** time includes copying data between kernel space and user space; the **mbcopy** row in Table 2 contains the time make copies of the data for retransmissions; and the **ATM** row includes the time spent copy data between the host and the device.

From this breakdown we see that data is copied at least twice on both sends and receives. The copy in **mbcopy** only occurs on sends, and is made from the mbuf chain for retransmissions. Eliminating the checksum (discussed in Section 4.2) opens the possibility of eliminating these data copying costs given a network adapter that supports DMA. With a combined copy and TCP checksum, Clark et al. discuss a network adapter design that eliminates the need for a second copy[4]. In a later section, we investigate how combining a copy and checksum affects latency using the ATM adapter.

2.2.4 Scheduling

The scheduling times (the sum of the **IPQ** and **Wakeup** rows) for switching contexts on the receive side are noticeable for small data transfers (68 μ s out of 1021 μ s, or 6.7% of the round trip time for the 4 byte case), particularly when compared to the costs of mbuf allocation and deallocation. However, scheduling costs do not contribute greatly to the overall latency of transferring large messages.

2.3 Measurement Summary

The detailed measurements have shown the contributions to latency of the various layers used in TCP communication. For large packet sizes, most of the overall processing time is spent in data copies and the checksum calculation significantly. For small packet sizes, the scheduling time and the time to do the TCP processing become noticeable when compared with the overhead of mbuf allocation and deallocation. However, for large transfers, the checksumming and copying data operations dominate the round trip times.

For the TCP layer in particular, the protocol processing time can be split into the time to perform the checksum, the time to do the copy during transmit, and the remainder. Although we do not further address the issue of the data copy, we address the problem of reducing the remaining protocol processing time using header prediction in the next section and the problem of optimizing the checksum in a subsequent section.

3 Header Prediction

Header prediction has often been suggested as a performance benefit for TCP[4]. There are two distinct kinds of optimizations that are often called header prediction. The first, involving prefilling parts of the transport header, is a known optimization for lowering latency[11, 15], and is not discussed further here. The second technique involves exploiting traffic locality to predict the next incoming packet to avoid the protocol control block (PCB) lookup cost. Others have studied using traffic locality to improve throughput for bulk data transfer protocols [2, 17]; we study its impact on latency.

In the BSD implementation, the TCP input processing engine keeps a single entry cache of the most recently used PCB. If the incoming packet is from the same connection as the previous packet, the call to the PCB lookup routine is avoided. The BSD 4.4 alpha TCP also precomputes the values it expects to find in the next incoming packet header, and can then execute a faster processing path if the prediction is correct. This notion is similar to the RPC "fast path" found in high performance RPC systems such as SRC RPC[15].

A related issue is the organization of PCBs, so that lookup is efficient in the case where there is a miss in the PCB cache. The insertion algorithm for the linked list of PCBs places the most recent creation at the head of the list. The lookup algorithm for the PCBs is just a linear search through the linked list of PCBs. McKenney and Dove

Size (bytes)	Round Trip Times (μ s)		Percentage Decrease (%)
	No Prediction	Prediction	
4	1110	1021	8
20	1127	1039	8
80	1324	1289	3
200	1560	1520	3
500	2186	2140	2
1400	2962	2976	0
4000	5950	5891	1
8000	11477	10636	7

Table 4

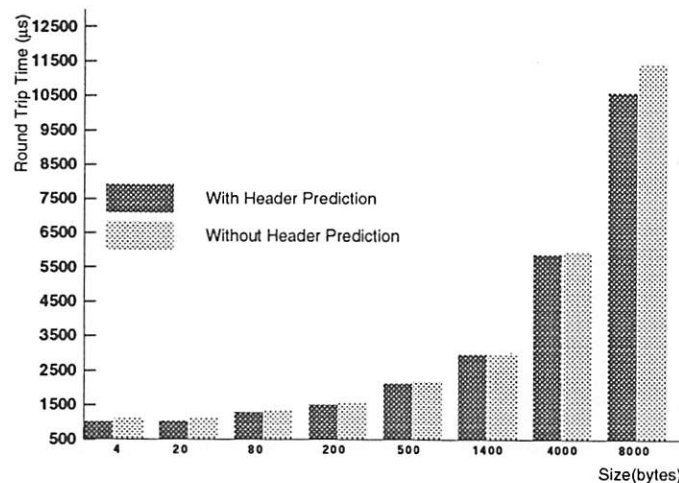


Figure 1

Effects of Header Prediction.

study alternative data structures for PCB lookup, and analyze these data structures by the expected average search length[12]. However, they do not discuss how long a search of any given length will take. While this facilitates comparisons, it is difficult to study the absolute effect of header prediction.

We measured the cost of a search for a variety of lengths, ranging from 20 entries (26μ s) to 1000 entries (1280μ s), and found that the results scaled linearly. The cost per element on a DECstation 5000/200 is just less than 1.3μ s. In addition, the typical number of active PCBs appears to be quite modest. For example, our departmental mail server has less than 250 active PCBs, and sampling thirty of our department workstations we found that all had less than 50. Given the relatively small memory requirements (even for 1000 PCBs), it seems that a simple hash table implementation could eliminate the lookup problem entirely.

In light of the above discussion, we decided to neglect the cost of the lookup and analyze the overall benefit of header prediction given that lookups are free. We built a kernel where both the PCB cache and the precomputation of the next incoming packet header (i.e. the TCP fast path) were disabled. By default, in our test environment, there will only be a very small number of TCP connections because our machines are only running the standard ULTRIX daemons and our test program.

Table 4 shows the results of this experiment, comparing a kernel with header prediction disabled to a kernel with it enabled. Figure 1 plots the same data graphically. For all the cases less than 8000 bytes, we notice only a very small improvement with header prediction, which is basically independent of data size. This small improvement is caused by a hit in the PCB cache, since the header precomputation and check (TCP fast path) fails in these cases

(as explained below). In the 8000 byte case, the larger difference comes from the header precomputation and check succeeding for half the received packets, as well as the hit in the PCB cache. The savings from the PCB cache hit are not large because the number of PCBs is small ($1.3 \mu\text{s}$ per PCB), and the TCP connection for our test program is likely to be near the head of the PCB list since recently created connections go at the head of the list. Even if there were many connections, a hash table implementation of PCBs would yield similar results.

The precomputation and check of the next header fails in all cases except the 8000 byte tests, where it succeeds half the time. In the 8000 byte case, this accounts for a small but noticeable difference. This is because two packets are being sent in the 8000 byte case, so the precomputation and check succeeds for the second packet. Upon closer inspection of the header prediction code, we discovered that the BSD 4.4 TCP header prediction only works in the two common cases of unidirectional data transfer. As the sender in a unidirectional transfer, header prediction succeeds when receiving an in-sequence acknowledgment with no data. As the receiver in a unidirectional transfer, header prediction succeeds when receiving an in-sequence data segment with no acknowledgment. Our test code creates the common case for a round-trip RPC style of communication where one receives data with a piggybacked acknowledgment, and this does not arise in a single sender, high throughput style of communication, which is what this code has been optimized for.

To summarize our results concerning header prediction, we found that the PCB cache accounted for a only a small improvement in latency (about 4% on average), and that the current implementation of header precomputation does not improve latency in a bidirectional RPC style of communication.

4 TCP Checksums

From the breakdown of the latency costs above, we see that, for large transfers, the cost of calculating the TCP checksum is a significant portion of round trip latency. In this section we introduce an optimized checksum algorithm and then discuss the kernel implementation issues of combining the checksum with a data copy. We then address the issue of eliminating the checksum for particular combinations of link types and applications.

4.1 Optimizing the Checksum

Others have noted that the ULTRIX 4.2A checksum algorithm could be improved by eliminating halfword accesses and using loop unrolling[9]. We implemented a similar optimized checksum algorithm; the performance of this algorithm and the ULTRIX algorithm at user level are shown in Table 5.

An optimization suggested in [3, 4] combines the checksum calculation with one of the data copies to eliminate redundant movement of data over the memory bus. In ULTRIX 4.2A, data is copied at least twice on both send and receive in addition to calculating the TCP checksum. One copy moves the data between user and kernel space, and the other copy moves the data between kernel and device memory.

We combined our optimized checksum algorithm with a data copy at user level to investigate its potential performance benefits. The results are shown in Table 5. The benefits are large: in the 8000 byte case, integrating the checksum and copy is 40% faster than performing the operations separately, and the effective bandwidth limitation imposed by the combined copy and checksum loop is just above 9 MB/s on the DECstation 5000/200.

The graph in Figure 2 shows the relative performance of the three methods for calculating the TCP checksum and copying the data.

We compare the performance of our implementation of the integrated checksum and copy with a user-level implementation on a Sun-3 described in [4]. Their measurements provide an interesting comparison of the scale in performance of a combined checksum and copy algorithm when changing hardware platforms. For example, with 1 KB of data they reported $130 \mu\text{s}$ to perform the checksum, and $140 \mu\text{s}$ to perform the memory to memory copy. The cost of their combined algorithm was $200 \mu\text{s}$. On the DECstation 5000/200, our optimized checksum takes $96 \mu\text{s}$ to checksum 1 KB of data, and the copy takes $91 \mu\text{s}$. The combined checksum and copy algorithm takes $111 \mu\text{s}$. The savings from the combined algorithm on the Sun-3 is 35%, and on the DECstation 5000/200 is 68%. The overall improvement when switching from the Sun to the DECstation is 80%.

Size (bytes)	Copy and Checksum Measurements (μ s)					Savings When Integrated (%)
	ULTRIX Checksum	ULTRIX bcopy	ULTRIX Total	Optimized Checksum	Integrated Copy and Checksum	
4	5	4	9	3	3	57
20	7	5	12	4	5	44
80	20	11	31	9	10	50
200	43	20	63	21	24	41
500	104	47	151	49	56	42
1400	283	124	407	134	153	41
4000	807	350	1157	378	430	41
8000	1605	698	2303	754	864	40

Table 5

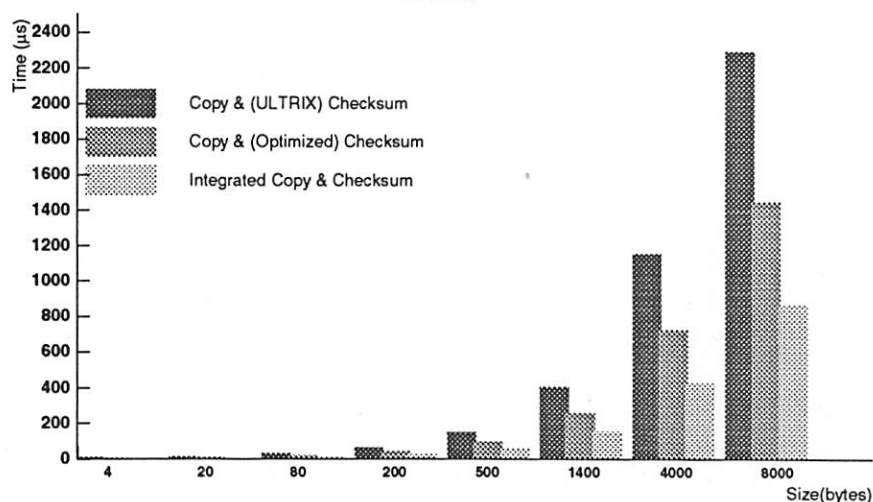


Figure 2

Copy and Checksum Measurements.

4.1.1 Kernel Implementation Issues

On the transmit side, the design of our ATM interface makes it impossible to defer the checksum calculation until the copy from kernel to device memory. Recall that it uses a simple memory mapped transmit FIFO. As soon as a single cell has been copied into the FIFO memory, the device begins to send it as later cells are still being copied to the device; there is no explicit action by the device driver to trigger the send. To compute the checksum, one must copy all of the data, and then write the checksum into the header of the packet. Therefore, it is impossible to combine the checksum and copy loops at the driver level given the FORE interface design.

Instead, we chose to integrate calculating the checksum during the copy from user to kernel space. The TCP checksum has the convenient property that one can calculate the checksums for pieces of a packet and then combine those partial checksums later. We calculate the checksum for each chunk of data copied into an mbuf at the socket layer, and store the partial checksum in the mbuf header. As long as all of the data in the mbuf are transmitted in the same TCP segment, then the TCP layer will not have to recalculate the checksum for that data. In our implementation, the socket layer chooses the amount of data to place in each mbuf independent of the current TCP segment size. One possible improvement to this scheme would be for the socket layer to predict future TCP segment sizes based on recent behavior. Another alternative would be to split the data in an mbuf into smaller chunks and calculate more than one checksum per mbuf, thus increasing the chance that a chunk will be transmitted in a single segment.

On the receive side, it will be difficult to postpone the checksum calculation until the kernel to user space copy because the protocol processing needs to know whether or not the incoming data is corrupt. Therefore, we have implemented the combined copy and checksum from the device memory to kernel memory. One disadvantage of this approach is that the device driver for each network interface needs to be modified to support this. Implementing the combined copy and checksum on the receive side is conceptually much simpler than on the send side because all the issues of mbufs and partial checksums disappear. However, the details of the implementation can be quite difficult and heavily dependent on the details of the device driver.

For comparison, the Digital OSF study also discusses implementing a combined copy and checksum in the kernel[3]. It appears that their combined copy and checksum is only used on the receive side for incoming UDP packets. Also, their implementation combines the checksum with the copy from kernel to user space, rather than from device to kernel memory. This requires that the user enable this code with a socket option because, in the case where the checksum fails, they must overwrite the user buffer with zeros to clear out the data that was just copied.

Size (bytes)	Round Trip Times (μ s)		
	Standard Checksum (μ s)	Combined Copy and Checksum (μ s)	Percentage Saving (%)
4	1021	1249	-22
20	1039	1256	-21
80	1289	1477	-15
200	1520	1707	-12
500	2140	2222	-3.8
1400	2976	2691	10
4000	5891	4644	21
8000	10636	8062	24

Table 6: Comparison of round trip latencies over ATM for the standard checksum and the combined copy and checksum calculation.

As an approximate measure of complexity, we added about 800 lines of code to implement the combined copy and checksum on both send and receive. The assembly language routines that implement the combined copy and checksum algorithm were less than half of the total number of lines of code, the rest was integration with the socket layer, the TCP layer, and the ATM driver.

The performance of our initial kernel implementation of the combined copy and checksum is shown in Figure 6. As expected, when the size of the data transfers increases, the combined checksum calculation provides significant savings in overall latency. In the 8000 byte case, the overall improvement has reached 24%. However, our initial implementation incurs significant costs in the smaller length cases, and the break-even point occurs somewhere between 500 and 1400 bytes.

4.2 Eliminating the TCP Checksum

The previous section has demonstrated that combining the checksum calculation with a data copy reduces latency. However, it is clear that latency can be further reduced by eliminating the checksum calculation altogether. It is already common practice to eliminate the UDP checksum for local area NFS traffic. Kay and Pasquale describe a mechanism using the Alternate Checksum Option to negotiate connections that do not use the checksum[8]. We therefore restrict ourselves to an analysis of the error characteristics and the implications of eliminating the checksum for local-area ATM traffic. We define local-area traffic as packets that go from source host to destination host without passing through any IP routers.

To measure the latency effect of eliminating the TCP checksum, we compare the round-trip times of the various packet sizes with and without the checksum calculation. Table 7 shows the results of eliminating the checksum on round trip measurements. The packet sizes are in bytes, and all times are in microseconds. The **Checksum** column shows the average round-trip latency when the checksum is calculated; **No Checksum** shows the average round-trip latency when the checksum is not calculated; and **Percentage Saving** is the relative saving when the checksum is eliminated. On the 4 byte case where the checksum overhead is minimal, nothing is gained. But, as the packet size increases, eliminating the TCP checksum significantly improves communication latency, e.g., the latency of the 8000

byte case is reduced by about 40%.

Size (bytes)	Average ATM Round Trip Time Without Checksum		
	Checksum (μ s)	No Checksum (μ s)	Percentage Saving (%)
4	1021	1020	0.1
20	1039	1020	1.8
80	1289	1233	4.3
200	1520	1392	8.4
500	2140	1808	16
1400	2976	2083	30
4000	5891	3633	38
8000	10636	6233	41

Table 7: Comparison of round trip latencies over ATM with and without the TCP checksum calculation.

With proper support from the host-network interface and the processor-memory subsystem, eliminating the TCP checksum can also benefit throughput oriented applications. For example, having DMA capability in the host-network interface and a snoop cache as found in [5], allows data to be moved at near bus bandwidth speeds to the application layer. In contrast, as Section 4.1 indicates, even an integrated copy and checksum routine limits bandwidth to about 9% of the bus bandwidth on the DECstation 5000/200.

4.2.1 System Issues in Checksum Elimination

Eliminating the TCP checksum on local-area ATM networks is a delicate system design decision and we discuss some of the relevant system-level issues first before discussing its performance implications.

The “end-to-end argument”, a classic principle in system design, says that the two ends of a reliable communication path should not depend on any of the intervening system components for correctness [14]. In other words, to assure the integrity of the communicated data, the communication end points must do a check independent of any checks done by intermediary components. If checks are done by intermediate or internal layers, they serve only as potential performance optimizations and do not subsume the end-to-end correctness check.

Intermediate checks can serve as performance optimizations, for example, by providing an inexpensive check that detects frequently occurring errors that would otherwise invoke a more expensive end-to-end recovery mechanism. On the other hand, an intermediate check can result in overall performance loss if, for example, it is expensive to perform and detects only infrequent errors; in such a case, even a very expensive end-to-end recovery could be preferable since the error seldom happens and the recovery cost can therefore be amortized.

In cases where TCP is used by a higher level service that performs its own checks, such as RPC systems that check their arguments, there is some debate on whether it is prudent to eliminate TCP checksums. The original environment that TCP was developed in used low-bandwidth links with little support for link-level error detection in hardware. Thus, the cost of detecting an error at the application layer impacted performance significantly. The use of the TCP checksum was therefore a performance optimization, consistent with the spirit of the end-to-end argument.

However, ATM networks with fiber optic links have very low error rates. For example, the bit error rate is on the order of 10^{-12} errors/s (i.e., one bit error in 3 hours if the network is used continuously at a bandwidth of 100 Mbits/s)[7]. Further, standard ATM adaptation layers (e.g., AAL3/4 and AAL5) specify *end-to-end* CRC checksums on the data, and host-network interfaces implement these in hardware. Thus, in cases where TCP is used as an intermediate layer to deliver data from an ATM network to a higher layer that will perform its own data integrity checks, eliminating the TCP checksum seems acceptable both on the grounds of performance and the end-to-end principle.

The main purpose of layering a TCP checksum over a link-level CRC is to potentially detect errors that the CRC does not catch. These errors can arise from four sources: (1) errors introduced by switches in transferring data between their input and output ports, (2) errors introduced by the network controllers in moving data between host and controller memories, (3) erroneous data injected into the network through external gateways or bridges, and (4) errors introduced by the link that are theoretically not detectable by the CRC because of the properties of the erroneous bit pattern and the CRC.

The first source of errors is not a problem since AAL payload checksums are end-to-end, i.e., intermediate switches do not recompute the checksum. The second type of errors is a potential problem, if for example, a buggy network controller introduces errors in transferring data between host and controller memory. We view this as a hardware problem in the controller that can be fixed using a hardware solution, e.g., incorporating parity or ECC into the controller memory.

The impact of the third type of errors can be eliminated by the application selectively using the TCP checksum elimination option only for local-area traffic. In fact, experiments conducted with and without wide-area traffic on our departmental Ethernet indicate that TCP detects two orders of magnitude fewer errors than the Ethernet CRC when wide-area traffic is included. Without wide-area traffic, TCP detected no checksum errors. We expect similar behavior on a local ATM network with quieter fibers. The quieter fiber also reduces the likelihood of errors from the fourth source.

We do not believe that eliminating the TCP transport entirely will be an option because the ATM network does not guarantee freedom from cell loss and some form of reliable transport mechanism will be required. The cost savings of optionally eliminating the TCP checksum, though, may be attractive to some applications. Other applications that are unable or unwilling to perform the necessary application-specific checks to guard against data corruption can continue to use the checksum at a cost.

4.2.2 Summary

To summarize, our measurements involving the TCP checksum suggest that there are definite performance advantages in making it optional. Further, for certain combinations of link types and applications, we believe it is possible to eliminate the TCP checksum without compromising error detection efficiency or violating established system design principles.

5 Conclusions

In this paper we have investigated the latency characteristics of TCP, and how optimizations originally proposed to improve throughput affect latency.

In previous experience with designing high performance "lightweight" RPC systems, we found that network driver and adapter design have a significant impact on performance[18]. For TCP, we also found that the network driver, adapter, and physical link have a significant impact on the latency.

In this paper, we first characterized the latency costs of TCP by breaking down round trip times for data transfers ranging in size from 4 bytes to 8000 bytes. We found that operating system services such as memory allocation contributed little to protocol processing time, particularly compared to the cost of scheduling. We also found that data-touching operations, such as copying and checksumming, dominate latency for transfers larger than 200 bytes.

We also found that header prediction had only a small impact on latency, primarily because the TCP fast path for input processing is not used for the round trip style of communication we measured.

We have found that computing the TCP checksum is a major cost of the overall TCP processing, and have characterized the effect of two optimizations to the checksum process. The first is an optimized implementation of the checksum algorithm that uses loop unrolling. The second optimization combines the optimized checksum calculation with copying the data. For large transfer sizes, the combined checksum and copy mechanism decreases round trip latency by as much as 24%.

In addition to optimizing the TCP checksum process, we have argued that, for particular combinations of link types and applications, it is possible to eliminate the TCP checksum calculation. Once the checksum is eliminated, round trip latency improves by as much as 41%.

6 Acknowledgements

We would like to gratefully acknowledge Ed Lazowska for his encouragement and comments on this project and report, as well as the helpful suggestions of the program committee. We also wish to thank Brian Bershad for his diligent shepherding that added greatly to the clarity of the paper.

References

- [1] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. "Lightweight Remote Procedure Call." In *ACM Transactions on Computer Systems*, 8(1):37-55, February, 1990.
- [2] John B. Carter and Willy Zwaenepoel. "Optimistic Implementation of Bulk Data Transfer." In *Proceedings of the 1989 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1989, pp. 61-69.
- [3] Chran-Ham Chang, Dick Flower, John Forecast, Heather Gray, Bill Hawe, Ashok Nadkarni, K. K. Ramakrishna, Uttam Shikarpur and Kathy Wilde. "High-Performance TCP/IP and UDP/IP Networking in DEC OSF/1 for Alpha AXP." *Digital Technical Journal*, Vol. 5 No. 1, Winter 1993, pp. 44-61.
- [4] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. "An Analysis of TCP Processing Overhead." *IEEE Communications Magazine*, June 1989, 23-39.
- [5] Digital Equipment Corporation, Maynard, MA. *Alpha Architecture Reference Manual*, 1992.
- [6] FORE Systems. *TCA-100 TURBOchannel ATM Computer Interface, User's Manual*, 1992.
- [7] Daniel H. Greene and J. Bryan Lyles. "Reliability of Adaptation Layers" *Protocols for High-Speed Networks, III*, Elsevier Science Publishers B.V., 1993, pp. 185-200.
- [8] Jonathan Kay and Joseph Pasquale. "A Performance Analysis of TCP/IP and UDP/IP Networking Software for the DECstation 5000" *Tech Report, CSL U.C. San Diego/Sequoia*, December 1992.
- [9] Jonathan Kay and Joseph Pasquale. "Measurement, Analysis, and Improvement of UDP/IP Throughput for the DECstation 5000" *Tech Report, CSL U.C. San Diego/Sequoia*, January 1993.
- [10] Van Jacobson, Robert Braden, and David Borman. "TCP Extensions for High Performance." RFC 1323, LBL, USC/ISI, and Cray Research, May 1992.
- [11] David B. Johnson and Willy Zwaenepoel. The Peregrine high-performance RPC system. *Software - Practice and Experience*, 23(2):201-221, February 1993.
- [12] Paul E. McKenney and Ken F. Dove. "Efficient Demultiplexing of Incoming TCP Packets." In *Proceedings of SIGCOMM '92*, August 1992, pp. 269-79.
- [13] John K. Ousterhout. "Why Aren't Operating Systems Getting Faster As Fast as Hardware?" In *Proceedings of the USENIX 1990 Summer Conference*, June 1990, pp. 247-256.
- [14] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277-288, November 1984.
- [15] Michael D. Schroeder and Michael Burrows. "Performance of Firefly RPC." *ACM Transactions on Computer Systems*, 8(1):1-17, February 1990.
- [16] Michael D. Schroeder, Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Satterthwaite, and Charles P. Thacker. Autonet: A high-speed, self-configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communications*, 9(8):1318-1335, October 1991.
- [17] Cheng Song and Lawrence Landweber. "Optimizing Bulk Data Transfer Performance: A Packet Train Approach." In *Proceedings of SIGCOMM '88*, September 1988, pp. 134-144.
- [18] Chandramohan A. Thekkath and Henry M. Levy. "Limits to Low-Latency Communication on High-Speed Networks." *ACM Transactions on Computer Systems*, 11(2):179-203, May 1993.

Alec Wolman is a graduate student at the University of Washington, currently on leave from Digital Equipment Corporation's Cambridge Research Lab. He holds an A.B. in Computer Science from Harvard University. His electronic mail address is wolman@cs.washington.edu.

Geoff Voelker is a graduate student at the University of Washington. He received the B.S. in Electrical Engineering and Computer Science from the University of California at Berkeley in 1992. His electronic mail address is voelker@cs.washington.edu.

Chandramohan A. Thekkath is a candidate for the Ph.D. degree in the Department of Computer Science & Engineering at the University of Washington. His electronic mail address is thekkath@cs.washington.edu.

Improving UNIX Kernel Performance using Profile Based Optimization

Steven E. Speer (Hewlett-Packard)

Rajiv Kumar (Hewlett-Packard)

and

Craig Partridge (Bolt Beranek and Newman/Stanford University)

Abstract

Several studies have shown that operating system performance has lagged behind improvements in application performance. In this paper we show how operating systems can be improved to make better use of RISC architectures, particularly in some of the networking code, using a compiling technique known as Profile Based Optimization (PBO). PBO uses profiles from the execution of a program to determine how to best organize the binary code to reduce the number of dynamically taken branches and reduce instruction cache misses. In the case of an operating system, PBO can use profiles produced by instrumented kernels to optimize a kernel image to reflect patterns of use on a particular system. Tests applying PBO to an HP-UX kernel running on an HP9000/720 show that certain parts of the system code (most notably the networking code) achieve substantial performance improvements of up to 35% on micro benchmarks. Overall system performance typically improves by about 5%.

1. Introduction

Achieving good operating system performance on a given processor remains a challenge. Operating systems have not experienced the same improvement in transitioning from CISC to RISC processors that has been experienced by applications. It is becoming apparent that the differences between RISC and CISC processors have greater significance for operating systems than for applications. (This discovery should, in retrospect, probably not be very surprising. An operating system is far more closely linked to the hardware it runs on than is the average application).

The operating system community is still working to fully understand the important differences between RISC and CISC systems. Ousterhout did a study in 1990 that showed that the failure of memory systems to keep pace with improvements in RISC processor performance has had a disproportionately large effect on operating system performance [1]. Mogul and Borg [2] showed that context switches in RISC processors take a surprisingly long time and suggested cases where busy waiting was actually more efficient than taking a context switch. Implementation work by Van Jacobson at Lawrence Berkeley Labs [3] (supported by work by Partridge and Pink [4]) has shown that techniques such as fitting the instructions to do an IP checksum into the otherwise unused instruction cycles between load and store instructions in a memory copy can dramatically improve networking performance. These studies are probably only the beginning of a series of improvements that will come about as operating systems designers better understand RISC platforms.

(We should note that newer CISC processors are increasingly using features such as pipelines and super-scalar architectures pioneered by RISC systems. As a result, many of the optimizations used on RISC processors now can be applied to kernels on CISC processors as well. However, to simplify discussion the rest of this paper will continue to talk of "RISC" systems).

This paper looks at another performance question in operating system performance on RISC systems. By their nature, operating systems spend a lot of time doing tests such as confirming that arguments to system calls are valid, that packet headers received from a network are valid, or that a device is idle (or busy). Ultimately, these logical tests are encoded as conditional branch instructions on the target processor.

The frequent tests have at least two causes. First, operating systems typically have multiple concurrent operations in progress at once, and must regularly check that the operations do not interfere. Second, an operating

system spends much of its time handling small requests from possibly defective (and thus untrustworthy) applications and therefore must confirm that any information it receives is correct. One can contrast this approach with programs, which rarely support concurrency and often do large amounts of computation without having to worry about external sources of errors.

Because operating systems have to do so much testing, we hypothesized that one reason operating systems do less well than applications on RISC systems is that they suffer more from branch penalties and cache misses. In other words, because operating systems have a lot of branches and because mispredicting which way a branch will go often causes a processor pipeline stall and even a cache miss, operating systems lose a lot of performance due to mispredicted branches. (For instance, in the HP-UX operating system [5] on the PA-RISC processor [6], nearly one instruction in five is a branch instruction, and almost half of those are conditional branches). In our work, we expected branch-related performance problems would be particularly notable in the networking code, because networking code must deal with both input from applications and input from the network. To test our hypothesis, we examined ways to tune the kernel to minimize mispredicted branches, with particular attention paid to performance improvements in the networking code.

2. RISC Processors and Profile Based Optimization

The technique we used to experiment with kernel performance is called profile based optimization (PBO). This section describes RISC architectures in brief to illustrate the problems PBO tries to solve and then explains PBO.

2.1. RISC Processors

RISC processors differ from other processor architectures primarily in the extreme uniformity enforced on the instruction set in order to create a simple instruction pipeline. Because our work was done on a PA-RISC processor (the PA-RISC 1.0, known as Mustang), this section will use that processor as the example.

The PA-RISC Mustang has the five stage CPU-pipeline as shown below:

PA-RISC Mustang CPU Pipeline				
Stage 1	Stage 2	Stage 3	Stage 4	Stage 5
Fetch	Inst Decode	Branch	Arithmetic	Results

The pipeline can be viewed as an assembly line where one new instruction is fed into the pipeline at the start of each clock cycle and moves one stage further through the pipeline with each clock tick. Five instructions can be processed concurrently in the pipeline. Instruction fetch takes place in the first stage, followed by instruction decoding. After the instruction is decoded, branch targets are calculated, logic operations are done, and conditions codes are set in the third and fourth stages. Finally, the results of the operations are written in the fifth stage.

If an instruction to be fetched in the first stage is not in the processor's instruction cache, the processor stalls. Execution resumes when the missing instruction is retrieved from main memory. Instructions are loaded from main memory in groups of 8 instructions (a 32-byte cache line). The processor can start executing the missing instruction as soon as it comes in from main memory (even if the cache load is not yet complete). The instruction cache on this machine is a direct mapped cache and distinct from the data cache.

One problem with pipelining is that for conditional branch instructions, the instruction must move through the first two stages of the pipeline before the processor can determine the value of the condition. The value of the condition determines which of two instructions (the instruction after the branch, or the target of the branch) will next be executed. Because the outcome is not known until the branch instruction is at the end of the third stage, if the processor is to keep the pipeline full, it must make a guess about the outcome of the condition. If the processor guesses wrong, it must nullify the instructions that have been incorrectly started down the pipeline and then start over, loading the correct instruction in the first stage of the pipeline. The cycles wasted due to mispredicted branches (those occupied by instructions that were fetched incorrectly and later nullified) are called control hazards or branch stalls. Their effect on execution time is roughly the same as inserting several NO-OP instructions into the executing programs instruction stream.

Another general problem with branches (conditional or not) is that when a branch is taken, its target instruction may not be in the processor's instruction cache, causing a processor stall (even more NO-OPs). To reduce the chance of a cache miss, it is usually desirable to have the most likely target of the branch be near the branch instruction so that the branch and its target are more likely to be cache resident at the same time.

A less obvious penalty for mispredicting a branch is that the mispredicted instruction may also cause a cache miss. This cache miss can be harmful in two ways. First, the code containing the mispredicted instruction may overwrite a section of processor cache that contains active code (which may have to be reloaded later). Second, if the correct instruction is also not in cache, loading the wrong 8 instructions may delay loading the correct instruction because the processor cache line is still busy loading the wrong instructions when the processor issues the request for the correct instruction. Thus the net effect of a mispredicted branch is often that processor resources are used very inefficiently [7].

Overall, therefore, it is reasonable to expect that minimizing the number of mispredicted branches has the potential to reduce the execution time of a program or operating system, both by reducing branch stalls and increasing overall instruction cache efficiency. At a minimum, each branch stall eliminated will reduce execution time by at least one clock cycle.

2.2. Profile Based Optimization

The work described in this paper employs a technique for improving branch prediction and instruction cache usage using profile information gathered from a running system and analyzed by the compiler. The technique is known as Profile Based Optimization or PBO [8]. Support for some PBO has been available in the HP-UX compilers since version 8.02. We used the HP-UX 9.0 compiler which is more aggressive in its use of profile data to do code optimization.

Although PBO has already been applied to several user level applications such as SPEC92, database servers, compilers and Xserver, it has not previously been applied to an operating system. We had to implement additional profiling support in the HP-UX kernel to make these experiments possible. (PBO requires kernel modifications for the same reason `gprof` [9] does. Unlike applications, the kernel does not load initialization routines from a library and does not normally terminate. So routines to cause the kernel to initialize PBO data structures and to retrieve PBO data from a running kernel were needed).

2.3. What PBO Does

The Profile Based Optimization available in version 9.0 of the HP-UX language products currently performs two optimizations. These optimizations are:

1. To reorder the basic blocks of a procedure such that the most heavily used basic blocks (a code sequence without a branch) are placed contiguously in an order which favors the hardware's branch prediction logic. This optimization reduces the overall number of mispredicted branches during execution.
2. To reorder the procedures of a program such that procedures on most frequent call-chains are laid out contiguously. By repositioning the procedures in this order, paging and to a lesser extent, cache misses, are reduced [8]. Also by placing most caller-callee pairs closer together in the code layout, fewer long branch sequences are needed to reach the callee (a long branch on PA-RISC takes two instructions).

Observe that both optimizations are code optimizations and nothing is done to reorder the data segment of the program. Furthermore, the two optimizations are independent of one another.

2.4. How PBO Works

PBO produces and uses an execution profile from a program to perform its optimizations in a three step process.

In the first step, the program code is instrumented to collect an execution profile. On the HP-UX C compiler, the `+I` option causes the code to be instrumented. Every arc connecting the nodes of a basic blocks control graph for each procedure is instrumented during the code generation phase. After code generation, the linker then adds

instrumentation for every arc of the procedure call graph.

The second step is to run the instrumented executable with some typical input data. For regular programs that run to completion, a profile database is generated when the program completes. In the kernel's case, a separate application must extract profile information from the kernel when profiling has been completed.

The last and the final step is to recompile the program using the profile data as an input to the compiler. Under HP-UX, the `+P` option tells the compiler to examine the profile data. The compiler reads the profile database and does basic block repositioning according to the profile. After the object files are built, linking is performed. The linker reads the same database and performs procedure repositioning as dictated by the profile.

3. Benchmark Results

To test the effects of PBO on the HP-UX kernel, we selected four benchmarks that spent a significant amount of their total execution times in *system* (as opposed to *user*) mode. The benchmarks chosen were:

1. *McKusick Kernel Performance Benchmark.*
This benchmark was developed to assist in the performance improvement of 4.2BSD UNIX at U.C. Berkeley in the 80's [10]. The benchmark was designed to be relatively insensitive to hardware configuration changes as long as its minimum needs are met. It performs a number of common system services with several different variations in system call parameters. In this case, the benchmark works heavily on `exec`, `pipes`, `fork` and other system calls. The overall time to execute the applications (rather than the kernel performance) is measured.
2. *OSBENCH Operating System Performance Benchmark.*
The OSBENCH benchmark was developed by John Ousterhout at U.C. Berkeley and used in the 1990 study [1]. It is designed to measure the performance of some basic operating system services. The benchmark performs memory copies with various size buffers, performs some rudimentary file manipulation using various sizes of files, tests pipe performance, checks simple system call response time using `getpid()`, measures read and write performance with a variety of parameters. It reports results for the specific micro benchmarks.
3. *The KBX Kernel Benchmark.*
This benchmark was developed by Doug Baskins at Hewlett-Packard to aid in the performance tuning of the HP-UX operating system on HP workstations. It is similar to the other kernel benchmarks in that it measures the time it takes to perform several iterations of various system requests.
4. *Netperf Network Performance Benchmark.*
Netperf was also developed at Hewlett-Packard by the Information Networks Division. It is a proprietary tool designed primarily to measure bulk data transfer rates and request/response performance using TCP and UDP. In our tests, we used it to measure the time it took to move data through the protocol stack in the kernel.

The first three benchmarks were chosen because they are generally well-known and understood kernel benchmarks. The `netperf` benchmark was chosen because it stressed the networking part of the kernel, which we thought might give particularly promising results.

The instrumented kernel was run with a single benchmark to obtain a profile database. The kernel was then recompiled with the profile database to produce an optimized version. Note that the kernel was optimized separately for each of the four benchmarks and the only optimization applied was PBO.

Once an optimized kernel was generated, it was run with the same benchmark and benchmark inputs used to generate the PBO database. The results of the optimized kernel were then compared to those obtained by running the base kernel that was built without any optimization. In the sections that follow, performance improvements are always calculated as:

$$(time_base - time_opt) / time_opt$$

where *time_base* is the time it took to execute the benchmark (or portion thereof) on the base kernel and *time_opt* is the time it took to execute the benchmark on the optimized kernel. We also report the standard deviation for the

optimized results measured.

The results reported were obtained on an HP 9000/720 workstation with 64MB of RAM. The machine was configured with 2 SCSI disks, each with approximately 100 MB of swap space, and connected to a 10 Mb/Sec Ethernet. The workstation was running Release 8.07 of the HP-UX Operating System.

3.1. McKusick Results

The McKusick test measures aggregate performance and was run for five iterations on each system (because the time for each iteration was small). The results were:

McKusick Benchmark Performance Results			
Mean Time Base System (sec)	Mean Time Optimized System (sec)	Std Dev	Improvement
117.40	112.00	0.89	4.8%

Note that because the McKusick benchmark measures application performance, the kernel improvement is diluted by application time spent in user space (which was not optimized). System time accounted for approximately 85% of execution time which hints that the OS performance improvement measured by this benchmark is around 6%.

3.2. OSBENCH Results

OSBENCH was run three times each on both the base system and the optimized systems. The results are summarized in the table below. (Cases where the standard deviation was larger than the measured improvement are marked with a '*').

OSBENCH Optimization Measurements				
Time Optimized	Time Base	Standard	Perf. Impr.	units measured
32.8	32.4	0.66	*	cwcor_1K milliseconds
32.2	32.6	0.12	1.1%	cwcor_4K milliseconds
32.2	32.3	0.24	*	cwcor_8K milliseconds
56.3	65.8	0.53	17.0%	cwcor_16K milliseconds
99.2	115.3	0.59	16.2%	cwcor_64K milliseconds
149.7	166.2	0.19	11.0%	cwcor_128K milliseconds
0.42	0.42	0.01	*	cswitch milliseconds
0.17	1.18	0.00	*	getpid microseconds per iteration
0.16	0.17	0.01	*	open_close milliseconds per open/close pair
30.095	31.697	0.443	5.3%	read megabytes/sec.
53.25	60.04	5.72	12.8%	select1/0 microseconds per iteration
56.09	58.33	0.13	4.0%	select1/1 microseconds per iteration
86.26	94.27	5.65	9.3%	select10/0 microseconds per iteration
87.14	92.07	0.31	5.7%	select10/5 microseconds per iteration
83.14	86.52	3.87	*	select10/10 microseconds per iteration
107.81	109.62	0.51	1.7%	select15/0 microseconds per iteration
109.4	110.8	5.90	*	select15/5 microseconds per iteration
102.6	102.8	2.41	*	select15/15 microseconds per iteration

The results were somewhat disappointing because any improvement in most of the micro benchmarks was obscured by the variation in the measured performance values.

3.3. KBX Kernel Benchmark

The KBX benchmarks proved to give far more useful results. Its micro benchmark results are shown below and are the results of five runs of the benchmark.

KBX Optimization Measurements				
Time Optimized	Time Base	Std Dev	Perf. Impr.	Microbench Measured
1.6	1.6	0	0%	getpid()
32.3	39.7	0.48	23%	ioctl(0, TCGETA, &termio)
19.8	19.2	0	-3%	gettimeofday(&tp, NULL)
1.0	1.0	0	0%	sbrk(0);
1.5	1.6	0.04	7%	j = umask(0);
402.6	416.7	1.25	4%	switch -- 2 × (if (read/write(fdpipe, BUF, 20) != 20))
139.0	141.2	0.32	2%	if (write/read(fdpipe, BUF, 20) != 20)
175.1	179.1	0.40	2%	if (write/read(fdpipe, BUF, 1024) != 1024)
425.2	441.9	1.39	4%	if (write/read(fdpipe, BUF, 8192) != 8192)
393.5	448.8	5.39	14%	j = creat("/tmp/temp", 0777) & close(j))
273.3	295.5	2.41	8%	pipe(pd); close(pd[0]); close(pd[1])
6327.6	6441.9	12.69	2%	if (fork() == 0) exit(0);
4321.2	4406.0	6.43	2%	if (vfork() == 0) _exit(0);
15067.3	15907.1	15.04	6%	if (fork() == 0) execve("kx", NULL, NULL);
16604.2	17409.9	68.15	5%	if (fork() == 0) execve("kx", args(1000), NULL);
15087.5	15901.6	31.00	5%	if (vfork() == 0) execve("kx", NULL, NULL);
16590.6	17411.8	36.72	5%	if (vfork() == 0) execve("kx", args(1000), NULL);
66.9	66.1	0.57	-1%	if (read(fd_file, BUF, 20) != 20) from cache
88.5	88.1	0.10	-.5%	if (read(fd_file, BUF, 1024) != 1024) from cache
230.4	231.2	0.59	.3%	if (read(fd_file, BUF, 8192) != 8192) from cache
99.4	106.8	0.45	7%	if (write/read(inet_socket, BUF, 20) != 20)
316.3	360.5	2.58	14%	if (write/read(inet_socket, BUF, 1024) != 1024)
1504.3	1622.8	53.75	8%	if (write/read(inet_socket, BUF, 8192) != 8192)
210.1	230.0	17.67	9%	if (write/read(unix_socket, BUF, 20) != 20)
228.9	241.3	3.76	5%	if (write/read(unix_socket, BUF, 1024) != 1024)
614.0	653.8	4.24	6%	if (write/read(unix_socket, BUF, 8192) != 8192)
22.78s	23.68s	0.27	4%	real time
3.28s	3.46s	0.15	5%	user time
19.17s	19.92s	0.11	4%	system time

With the exception of the huge improvement in *ioctl* performance (which was rather startling), the benchmarks generally suggest that major performance improvements came in tests that did context switching. Most of the read commands from files achieved little or no improvement, but activities that created a new context (e.g., *fork*) or sent data between processes (the socket and pipe tests) generally show a 5% to 10% performance improvement.

Some of the tests that make up this benchmark actually took slightly longer to run after being optimized than they did in the base case. This is because the optimization took data from the entire benchmark (not individual micro benchmarks) to generate a new operating system and the best performance for the entire benchmark sometimes forced blocks that did not get used as much into less advantageous positions relative to the arbitrary locations that they occupy in the base case. This observation is a strong reminder of the importance of selecting profiling inputs that mimic the software's most likely uses. In extreme cases, failing to do so could actually produce a system that would run slower overall after being optimized.

3.4. Netperf Network Performance Benchmark

Netperf can run a variety of benchmarks. The benchmark we used measures the performance single byte transactions (sending one byte messages which get one byte replies) using UDP and TCP over the loopback interface in the kernel. This benchmark was designed to measure the impact of PBO on the protocol processing (e.g., code to handle the protocol headers) portion of the networking code by minimizing the data in each packet. To avoid performance effects due to one-behind caches in the operating system, this benchmark was run several times, with increasing numbers of parallel senders and receivers. These results are the averages of 3 runs sending over the loopback interface. Performance is measured in bytes sent per second, but since each byte was sent in a single packet and received a single byte ack, the transmission rate is equivalent to one half the packets per second processed.

netperf Optimization Measurements Loopback Interface					
Number of Processes	TCP/UDP	Transmission (b/s) Base	Transmission (b/s) Optimized	Std Dev	Performance Increase
1	TCP	1568.13	2115.69	26.68	35%
2	TCP	1592.89	2124.56	39.87	33%
4	TCP	1556.98	2100.17	12.13	35%
8	TCP	1492.07	2001.53	9.65	34%
16	TCP	1459.67	1879.47	8.00	29%
1	UDP	1801.68	2164.94	10.03	20%
2	UDP	1800.52	2158.79	45.76	20%
4	UDP	1761.23	2164.87	25.44	23%
8	UDP	1685.63	2055.01	14.47	22%
16	UDP	1609.51	1951.92	8.27	21%

While the results do show that TCP, which has a number of internal caches, improves less as the number of processes increases, the transaction rate still improves sharply for both protocols, even when the number of parallel processes is large.

As an experiment, we then ran the test to a remote (unoptimized) host over an otherwise quiet Ethernet. This test is less representative of the effects of PBO, because the performance of both the remote host and the Ethernet affect the results, but it gives a feel for how much an optimized kernel might benefit from dealing with unoptimized kernel. Keep in mind as well, that while the kernel loopback is generally reliable and does not lose packets, real networks do lose packets so the transmission rates may be influenced by the need to retransmit a few packets.

Isolated Network Link to Remote Host					
Number of Processes	TCP/UDP	Transmission (b/s) Base	Transmission (b/s) Optimized	Std Dev	Performance Increase
1	TCP	1070.51	1204.96	4.86	13%
2	TCP	1821.14	2079.58	2.05	14%
4	TCP	1831.10	2081.56	19.97	14%
8	TCP	1838.91	2072.57	11.79	13%
16	TCP	1866.70	2109.90	12.31	13%
1	UDP	1163.50	1261.34	4.72	8%
2	UDP	1966.01	2064.20	4.81	5%
4	UDP	2011.84	2091.83	16.57	4%
8	UDP	2023.67	2094.12	18.34	3%
16	UDP	2052.65	2141.47	14.63	4%

The performance improvements are clearly far less than those achieved over the loopback interface in an optimized kernel, but a 14% improvement in TCP performance is still quite useful.

4. Conclusions

The results of the different benchmarks vary from a modest few percent to dramatic (over 30% in parts of the networking code), but the overall results are certainly encouraging.

The results imply that, using PBO, users can at tune their UNIX kernels to better serve the particular load patterns they experience without changing a line of source code. Obviously it is still important to replace slow kernel code with code that runs faster. The point is that PBO adds a final, finishing, level of optimization that can be very useful. Indeed, in some situations, such as the TCP and UDP transactions tested with *netperf*, the PBO optimizations can yield exceptional performance improvements.

Biographies

Steven Speer received his BS in Computer Science from Oregon State University and MS in Computer Science from Stanford University. He works for Hewlett-Packard in Ft. Collins, CO., in the division responsible for the operating systems on PA-RISC platforms.

Rajiv Kumar received his BS in Electrical Engineering from Indian Institute of Technology, Kharagpur and MS in Computer Science from University of Oregon. He works for the Hewlett-Packard Company with the group responsible for building optimizers for the PA-RISC architecture.

Craig Partridge received his A.B. and Ph.D. from Harvard University. He works for Bolt Beranek and Newman, where he is the lead scientist of BBN's gigabit networking project. He is an consulting assistant professor at Stanford University, Editor-in-Chief of *IEEE Network Magazine*, and the author of *Gigabit Networking*.

References

1. J.K. Ousterhout, "Why Aren't Operating Systems Getting Faster as Fast as Hardware?," *Proc. 1990 Summer USENIX Conf.*, Anaheim, June 11-15, 1990.
2. J.C. Mogul and A. Borg, "The Effects of Context Switches on Cache Performance," *Proc. 4th Intl. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Santa Clara, 8-11 April 1991.
3. V. Jacobson, *Tutorial Notes from SIGCOMM '90*, Philadelphia, September 1990.
4. C. Partridge and S. Pink, "A Faster UDP," *IEEE/ACM Trans. on Networking*, vol. 1, no. 4, August 1993.
5. F.W. Clegg, et al, "The HP-UX Operating System on HP Precision Architecture Computers," *Hewlett-Packard Journal*, vol. 37, no. 12, December 1986.
6. J.S. Birnbaum and W. S. Worley, "Beyond RISC: High-Precision Architecture," *Hewlett-Packard Journal*, vol. 35, no. 8, August 1985.
7. S. McFarling, "Program Analysis And Optimization For Machines With Instruction Cache," *Tech Report, Computer Systems Laboratory, Stanford University, Stanford CA*, September 1991.
8. K. Pettis and R.C. Hansen, "Profile Guided Code Positioning," *Proc. SIGPLAN on Programming Language Design and Implementation (SIGPLAN Notices)*, vol. 25, no. 6, June 1990.
9. S.L. Graham, P.B. Kessler, and M.K. McKusick, "gprof: A Call Graph Execution Profiler," *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pp. 120-126, June 1982.
10. S. Leffler, M. Karels, M.K. McKusick, "Measuring and Improving the Performance of 4.2BSD," *Proc. 1984 Summer USENIX Conf.*, Salt Lake City, June 12-15, 1984.

Memory Behavior of an X11 Window System

*J. Bradley Chen
School of Computer Science
Carnegie Mellon University*

Abstract

We used memory reference traces from a DEC Ultrix system running the X11 window system from MIT Project Athena and several freely available X11 applications to measure different aspects of memory system behavior and performance. Our measurements show that memory behavior for X11 workloads differs in several important ways from workloads more traditionally used in cache performance studies. User instruction cache behavior is a major component in overall memory system delays, with significant competition within and between address spaces. User TLB miss rates are up to a factor of two higher than other ill-behaved integer workloads. Write-buffer stalls, data cache behavior, and uncached memory reads can be problematic for microbenchmarks, but they are not an issue for the realistic applications we tested.

1. Introduction

We have used memory reference traces from DEC Ultrix, the X11 window system from MIT Project Athena, and freely available X11 applications to explore several aspects of memory system behavior and performance for X11 workloads. We measured behavior within the system, server and client, as well as interaction between address spaces.

Our analysis shows that memory behavior for X11 workloads differs substantially from that of traditional workloads, particularly in the instruction cache and TLB. Competition within and between contexts in the instruction cache has significant performance impact. This cache competition appears difficult to avoid in a direct mapped cache, suggesting that higher associativity may be required. TLB designs that do not accommodate the demands of large interactive systems may also become performance problems.

X11 workloads, as compared to the SPECmarks [23] and other more traditional workloads for behavioral studies of memory systems, differ in several fundamental ways:

- **Large program text.** Even the largest SPECmarks are small compared to X11. At 688 KBytes, *gcc* stands out among the SPECmarks for its large text segment¹. X11 servers commonly have as much as 1.8 megabytes of text, more than twice that of *gcc*. X11 clients also tend to have large code. The two real-world X11 clients used in this study, *gs* and *splot*, have text sizes of 946 Kbytes and 278 Kbytes respectively. User text size for *gs* with the X11 server is over four times that of *gcc*.

¹Text sizes are given for Ultrix DECstation executables. *gcc* is as built from the SPECmark distribution. The X11 server size is for */usr/bin/Xws* on an Ultrix workstation, which includes a number of DEC extensions. The tracing experiments used a smaller server (958 KBytes). See Section 3 for details on the server used in the tracing experiments.

This research was sponsored in by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597, and by an equipment grant from Digital Equipment Corporation.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Digital Equipment Corporation or the U.S. Government.

- **Three interacting contexts.** Typically, batch-oriented workloads involve two contexts: the user application and the kernel. For many of these workloads (scientific workloads are the most common examples) kernel activity is negligible. In contrast, activity in most X11 workloads is split among three contexts: the client, the X11 server, and the operating system, with significant activity occurring in all three contexts. The result is additional resource competition that does not happen in the two-context and single-context case.
- **Mandatory and potentially frequent context switches.** When multi-task workloads are used in memory system studies, they are usually created by taking unrelated batch-oriented workloads and running them simultaneously. Context switches for these multi-task workloads can often be scheduled arbitrarily. An intelligent scheduler may try to make switches infrequent, as a strategy for minimizing cache competition. In contrast, scheduler policy is irrelevant in client-server systems. Context switches are largely determined by client behavior and inter-process communication implementations. Depending on the client, context switches may be frequent.
- **A large community of users.** The X11 server and clients are used daily and repeatedly by a large contingent of the workstation computing community. Many of these users make rare use of programs such as those in the SPECmarks.

Performance for benchmarks is typically measured in terms of *throughput*, with program execution times reduced to units such as MIPS or MFLOPS. A key distinction between interactive workloads and more traditional benchmarks is their sensitivity to *latency*, which is the time required for the system to respond to a given input event. Analysis of memory system components such as caches and write buffers is common practice for throughput benchmarks [7, 8, 10]. However, interactive programs and client-server systems have received relatively little attention in recent research. [3, 20]. This is unfortunate in that, for many computer users, quick response time for latency-critical interactive applications is more important than the throughput of batch jobs. Because of the size and complexity of server-based systems such as X11, few detailed measurements of their behavior have been made. We think the problem deserves more attention, as memory system delays can have a significant impact on latency for interactive workloads.

1.1. Related Work

This research focused primarily on measuring the behavior and performance of realistic X11 client workloads from the perspective of the memory system. Several prior studies measure X11 behavior, although they differ substantially in that they consider behavior at higher levels of abstraction. Researchers at the Microelectronics Computation and Technology Corporation built a tool called XSCOPE to measure X11 performance and localize performance problems [22]. XSCOPE provides information about X11 request, reply, error, and event packets. Their experience in designing XSCOPE indicated some problems with the syntax of the X11 protocol.

Simple measures of performance, such as operations per second, are often used when characterizing new graphics hardware. Researchers at DEC WRL have done significant work in achieving good X11 performance, both with simple bit-mapped framebuffers [16] and more complicated hardware [17]. They also demonstrate software algorithms that permit effective use of the hardware. They consider memory reference behavior, but strictly as related to frame buffer references; application performance is beyond the scope their work. Researchers at Hewlett Packard used a technique called Direct Hardware Accesses (DHA) in their Starbase/X11 Merge system to enable high performance when Starbase applications access the display. [6, 5]

Several other projects consider memory system performance in a more general context, independent of X11 applications. MTOOL [11] compares execution time of program segments to predicted time for a perfect memory system. A large difference between the predicted and the measured times suggests a possible memory system performance problem. MTOOL has been applied primarily to detecting memory bottlenecks in FORTRAN programs, and is not appropriate for measuring operating system behavior. Thus, MTOOL is not appropriate for measuring X11 workloads. MTOOL has been adapted to work with shared-memory multiprocessor programs [12]. Another project, MemSpy [15], is based on the Tango [9] simulation and tracing system. To date, Tango is designed for use with parallel applications and multiprocessor systems, and has not been applied to multiprogrammed uniprocessor workloads or measurements of operating system activity. The measurements for this paper concern aggregate memory system behavior. In contrast, both MTOOL and MemSpy identify performance problems as specific segments of code (also data for MemSpy) within a workload.

The remainder of the paper is organized as follows. The next two sections describe the experiments, first giving details on the tracing and simulation systems, then a qualitative and quantitative characterization of the workloads. Next, in the analysis section, we analyze memory delays for X11 workload from three points of view: memory penalties by subsystem, cache effects, and TLB behavior. The paper closes with a brief review of our major conclusions.

2. Tracing and Simulation

The experiments for this study ran on a DECstation 5000/200, using an address tracing system developed at Carnegie Mellon University and DEC WRL [4, 7]. The tracing system uses object code rewriting [4, 24], in which original object code is augmented with instrumentation instructions such that an address trace is generated as a side effect of program execution. Traces are accurately interleaved both within a single context and across user and system contexts. Traced addresses are corrected to reflect those of the original and not the traced instruction stream. The Ultrix kernel, the X11 server, and X11 clients were all instrumented and traced.

The DECstation 5000/200 uses a 25 MHz MIPS R3000 CPU with MIPS R3010 floating point unit and MIPS R3220 memory buffer. The DS5000/200 uses a simple eight-bit frame buffer interface, in which the frame buffer appears as memory and is written directly by the processor. No special-purpose graphics hardware is used. The X11 server runs as a user process. Communication between X11 server and clients occurs through the socket interface provided by Ultrix. The frame buffer is mapped directly into the address space of the X11 server, and accesses to the frame buffer bypass the cache. This could potentially induce penalties for frame buffer reads. Fortunately such reads are relatively rare. Frame buffer writes pass through the write-buffer so their performance is unaffected. A discussion of effective software support of the DECstation 5000/200 frame buffer can be found in [16].

instruction cache: 64 KB, direct-mapped, physical, 16 byte line, 15 cycle miss penalty.

data cache: 64 KB, direct-mapped, physical, 4 byte line, write allocate, 15 cycle read miss penalty, read miss fetches 16 aligned bytes.

write buffer: six entries, page-mode writes complete in one cycle, non page-mode writes complete in five cycles; CPU reads have priority for memory access, but wait for writes that have already started. 4 KB page size for page-mode writes.

translation buffer: 64 entries, 56 random/8 wired entries, trap to software on TLB miss. Each TLB entry maps a 4 KB page.

page mapping: Deterministic. The physical page used to back a given virtual page is determined by the virtual page number and its address space identifier. The deterministic strategy prevents conflicts within any 64 KB (cache size) range of virtual addresses.

kernel memory: All kernel text and most kernel data is in unmapped, cached physical memory.

Table 2-1: Memory system simulation parameters.

We used a DECstation 5000/200 memory system simulator, along with several other simple tools, to process trace generated by the test workloads. The parameters for the memory system simulation are given in Table 2-1.

We omit several million instructions from the start and end of each simulator experiment to eliminate startup and shutdown effects. The tracing system extracts page table information from the running kernel to provide virtual to physical page mappings in the simulator. The Ultrix page allocation code attempts to assign physical pages such that virtual page orderings are preserved in the physical cache. Kernel text is not mapped.

3. Workloads

We used the standard X11R5 distribution from MIT Athena (available by anonymous ftp from `export.lcs.mit.edu`). The PEX extension² was omitted from the X11 server. Otherwise, we used the default server configuration. The Ultrix system was version 4.2 revision 96. Table 3-1 describes the X11 clients used in this study. All programs are written in C, and compiled with version 2.1 of the DEC/MIPS C compiler.

workload	Description	time
micro benchmarks		
<i>destroy</i>	window destruction, using <i>x11perf-repeat 5 -reps 10 -subs 10 100 -destroy</i>	2.6
<i>resize</i>	window resize, using <i>x11perf-repeat 2 -reps 5 -subs 10 100 -resize</i>	2.5
<i>circulate</i>	window circulate operations, using <i>x11perf-repeat 2 reps 5 subs 10 100 -circulate</i>	2.8
<i>ftext</i>	text painting, using <i>x11perf-repeat 5 reps 500 -ftext</i>	2.4
<i>copy</i>	bitmap copy, using <i>x11perf-repeat 5 reps 250 -copywinwin100</i>	11.4
<i>scroll</i>	window scrolling, using <i>x11perf-repeat 2 reps 250 -scroll500</i>	23.3
X11 clients		
<i>splot</i>	Splot is run four times on four different input files. Total size of splot input is 94 Kbytes.	12.4
<i>gs</i>	Ghostscript is used to preview a twenty page conference paper. Input file size is 251 Kbytes.	25.9
Other workloads		
<i>gcc</i>	The GNU C compiler converts a 17K (preprocessed) source file into optimized Sun-3 assembly code. Not an X11 client.	3.7
<i>compress</i>	Data compression using Lempel-Ziv encoding. A 100K file is compressed then uncompressed.	1.3

Table 3-1: Experimental workloads. Times are in seconds.

X11perf is a client in the X11R5 distribution. It measures the time to repeat a given server operation some number of times, and is commonly used as a gauge of X11 server performance. All the microbenchmarks for this paper are runs of *x11perf* with different input parameters. *Splot* is a program for generating plots for PostScript and X11. It is available by anonymous ftp from `labrea.stanford.edu`. *Ghostscript*, by

²PEX is the PHIGS extension to X11, used for three-dimensional graphics.

Aladdin Enterprises, is an X11 previewer for Adobe Systems' PostScript language. It is distributed with the GNU General Public license, available by anonymous ftp from `athena-dist.mit.edu`. Version 2.6.1 of `gs` was used.

workload	instruction reads					data reads				data writes			
	idle	non-idle	%sys	%Xs	%Xc	non-idle	%sys	%Xs	%Xc	non-idle	%sys	%Xs	%Xc
destroy	0	33999	5.2	91.7	3.1	7128	5.5	91.1	3.4	4529	7.1	87.9	5.1
resize	0	35999	2.6	96.9	0.5	7382	2.2	97.2	0.6	2759	2.8	96.1	1.1
circulate	0	45999	2.9	96.7	0.4	9958	2.2	97.4	0.4	4284	3.4	95.9	0.6
ftext	0	62000	2.3	96.1	1.7	7921	4.2	92.5	3.3	4827	4.9	92.0	3.1
copy	0	92000	3.8	95.6	0.6	15912	3.4	95.5	1.1	14457	1.4	97.7	1.0
scroll	0	105999	2.0	97.9	0.1	32528	1.2	98.7	0.0	31740	0.5	99.5	0.0
splot	6910	148619	33.4	36.0	30.6	30852	32.2	34.9	32.9	16346	41.6	32.7	25.7
gs	3992	448018	10.0	30.8	59.2	93584	11.0	24.6	64.3	50994	14.6	33.4	52.0
gcc	63684	28899	21.0	0.0	79.0	5716	20.5	0.0	79.5	3810	28.6	0.0	71.4
compress	5555	16934	19.2	0.0	80.8	3249	18.9	0.0	81.1	2225	29.2	0.0	70.8

Table 3-2: Instruction and Data Reference Counts

This table shows event counts for each workload, along with the percentage contribution from the system (%sys), X11 server (%Xs), and X11 client (%Xc). The first column for each workload shows the number of idle instructions executed during that workload. All other counts and percentages are for non-idle events. All counts are in thousands.

Two workloads which are not X11 clients, *gcc* and *compress*, are included for comparison between X11 clients and other workloads. Among common integer benchmarks, *compress* presents an above average demand on the memory system, while *gcc*'s demands are extreme. More details on the memory system behavior of these workloads can be found in [7].

Table 3-2 gives reference counts for the experimental workloads. The low percentage of kernel instructions for the microbenchmarks demonstrates that many types of X11 operations require relatively little kernel activity. Higher levels of kernel activity in *splot* and *gs* are attributed (at least partially) to the file I/O that these workloads require.

4. Experiments and Analysis

4.1. Memory Cycles Per Instruction

We use data from our simulator to calculate *memory cycles per instruction*, (*MCPI*), which is the total number of CPU stall cycles due to the memory system divided by the total number of instructions executed. *MCPI* is one of several components of cycles per instruction (*CPI*), a metric commonly used to evaluate computer systems [14]. Other components of *CPI* that are not reflected in *MCPI* include one cycle per instruction for instruction execution by the processor, cycles during interlocked multiply, divide, and floating point operations, and no-ops inserted by the compiler for load and branch delays. The other components of *CPI* remain relatively constant as processor cycle time decreases. In contrast, *MCPI* is a function of the ratio of memory speed to processor speed, and is less dependent of processor architecture. *MCPI* will dominate overall *CPI* if current trends in processor and memory speed continue. *MCPI*+1 is a lower bound and a good estimate of overall *CPI* for workloads (such as operating systems) in which multiplies, divides and floating-point operations are rare.

Figure 4-1 illustrates *MCPI* for the experimental workloads. For comparison, *MCPI* measurements for *gcc* and *compress* are also shown. Each bar is shaded to denote different *MCPI* components. For visual clarity, the system and user contributions are separated by a vertical bar. Data and instruction cache misses in user and system mode are only partially responsible for the total *MCPI*. Other components include CPU write-stalls and uncached memory reads. CPU write-stalls are reflected in the *wbuffer* component which show the average per-instruction penalty from writes to a full write-buffer, and reads during the completion of a five-cycle write.

TLB misses are not included explicitly in the baseline *MCPI*. Their cost appears as additional instructions executed and additional data references, included in the total counts.

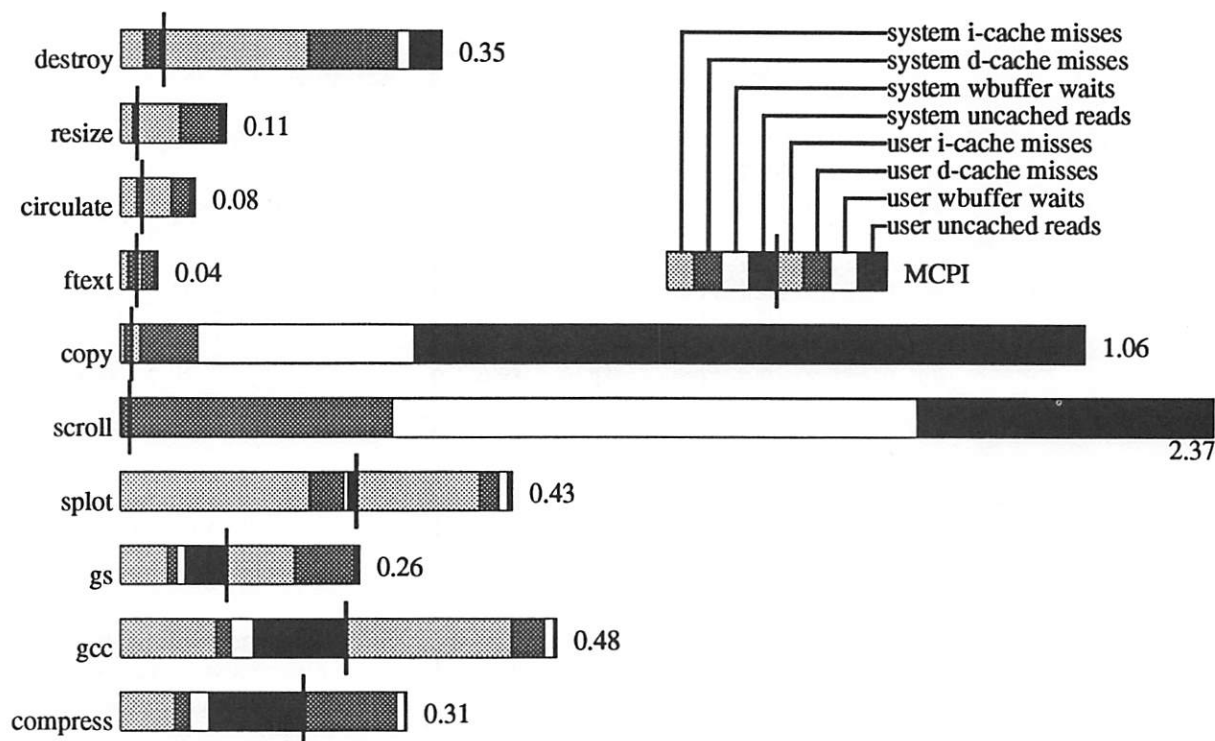


Figure 4-1: Baseline *MCPI*.

Each horizontal bar represents total *MCPI* for a given experimental workload, broken down between system/user and various components of the memory system. Contribution from system and user activity are separated by a vertical line. User activity includes X11 server and X11 client. The number at the right of each bar is the *MCPI* for that workload. Startup and shutdown effects were excluded by omitting several million instructions at the beginning and end of each simulation experiment. *MCPI* for *gcc* and *compress* are included for comparison with workloads that are not X11 clients.

As can be seen from Table 3-2, operating system overhead is low for the six microbenchmarks. This is reflected in Figure 4-1 as low system *MCPI*. With the X11 server accessing the frame buffer directly, kernel activity during the microbenchmarks is dominated by TLB faults and socket communication, both of which are relatively inexpensive as compared to the activity of disk I/O intensive workloads. In contrast, user-level *MCPI* varies significantly across the microbenchmarks, and is dependent on server activity. The worst behavior by far occurs for *copy* and *scroll*, which incur substantial write-buffer and uncached-read penalties due to a high density of frame-buffer references.

Comparing system behavior for the microbenchmarks to that of *splot* and *gs*, there is substantial additional system overhead for the realistic benchmarks. This reflects greater variation in system activity due to the addition of file I/O, and is consistent with behavior observed for system-intensive and I/O intensive applications such as *gcc*.

Turning to user-level overhead, Figure 4-1 shows that many X11 clients have significant user instruction cache *MCPI* contributions, sometimes higher than system i-cache *MCPI* contributions. This is unusual for integer workloads³ [7]. In the next section we discuss how competition within and between address spaces contributes to poor instruction cache behavior, for both system and user.

³*gcc* is exceptional in this respect.

Penalties from the write-buffer and uncached memory reads appear problematic in microbenchmarks but aren't significant in more realistic workloads. For frame-buffer writes, the X11 server benefits from the combination of write-buffer and writes through the uncached segment. Together they permit frame buffer writes to proceed at top speed without disturbing the contents of the data cache.

4.2. Cache Effects

We consider two types of cache misses:

- *Inter-Context Competition* occurs when references from two or more active address spaces displace each other in the cache. Client-server systems such as X11 introduce a user level server context, in addition to the application and system context of a workload such as *gcc*. The additional server context could induce more competition in the cache.
- *Self-Interference* misses occur when two active instructions in the same address space collide in the cache. The Ultrix page-mapping algorithm ensures that self-interference misses will not occur within an address space if a program's text is smaller than the cache size. Many of the SPECmarks have text smaller than the cache, but the text of X11 server, *gs* and *plot* are all much larger. With localities spread throughout large text, self-interference misses are more likely to occur.

We discuss inter-context competition first.

4.2.1. Inter-Context Competition

An X11 workload is composed of three contexts: X11 client, X11 server, and the operating system. Inter-context competition occurs when two or more contexts compete for memory resources (such as cache lines), causing degraded performance. To simulate a system without competition, we ran experiments with memory reference trace from only one source (eg. use trace from the kernel only), then summed events over all three runs. The sum represents the behavior of a hypothetical machine where each context has its own private memory system, hence a system where all competition has been eliminated. Figure 4-2 compares *MCPI* for several workloads with and without competition.

Figure 4-2 shows that, for realistic X11 workloads, inter-context competition has a large impact on overall *MCPI*. For both *plot* and *gs*, competition is responsible for a large proportion of system i-cache misses. Additionally, eliminating competition from *plot* causes a drastic reduction in user i-cache miss rates. The instruction cache requirements of *destroy* microbenchmark are modest, so eliminating competition has little effect. Similarly, *compress* shows little benefit when competition is eliminated.

We compared missing instruction addresses in the X11 server to counts of kernel instruction references for a run of *plot* to identify probable inter-context conflicts. An example of such a conflict was between the Ultrix general exception handler, `exception()`, and the X11 server memory allocation routine, `malloc()`. Both routines are called frequently, and are located in such a way that they overlap on a 4 Kbyte memory page. Kernel text pages are not mapped, so `exception()` will always be located in the same place in the cache. User text is mapped, so the location of `malloc()` in the cache will depend on the virtual-to-physical page assignment. In Ultrix the page assignment is a function of virtual page number and process ID. Given the page-mapping policy and cache parameters for this memory system, the probability that the above conflict will occur is 1/16. Many such conflicts were identified for the *plot* run. The large working sets of X11 workloads, combined with frequent mandatory context switches, makes competition misses more likely.

These results confirm earlier research on cache competition, which demonstrated significant penalties for warming up the cache after a context switch [19]. The earlier study found competition to be important in multitasking and compute-bound workloads, but a non-issue in the client-server workload they tested. However, the earlier study did not include system behavior, and it used a synthetic client-server workload dominated by communication, a workload more similar to the microbenchmarks used for this study than the realistic workloads. Our work complements the prior study by including system effects in the memory reference stream, and by demonstrating that inter-context competition can also have impact for client-server workloads.

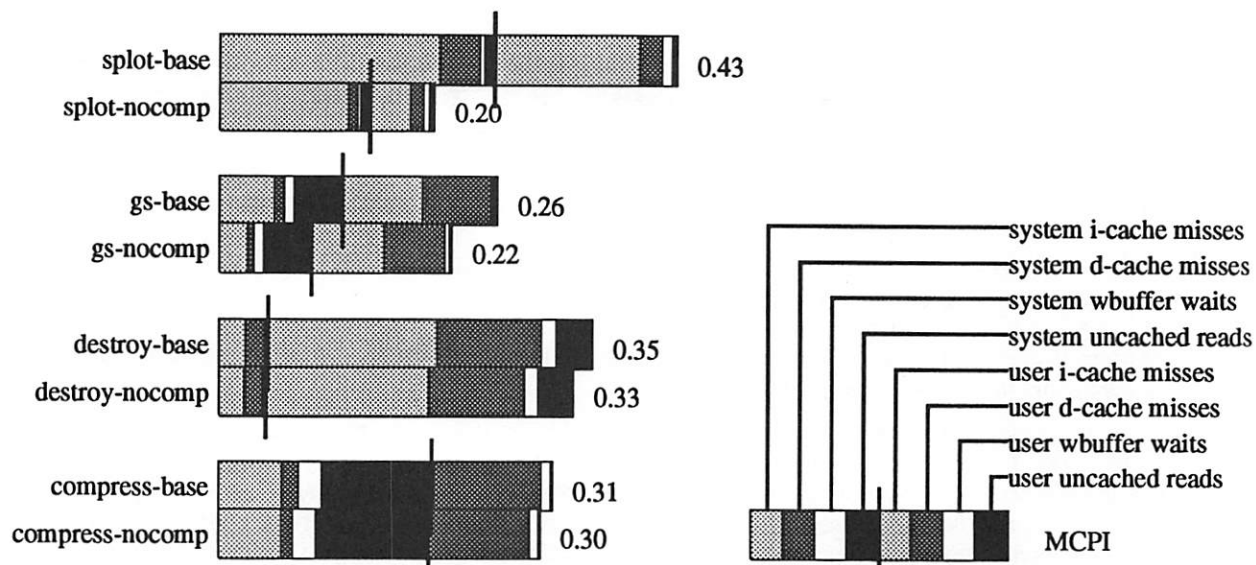


Figure 4-2: MCPI with and without inter-context competition.

This figure shows the effect of inter-context competition on memory system performance. Each horizontal bar represents total MCPI for a given experimental run, as in Figure 4-1. We show two bars for each workload, the upper corresponding to the base system, and the lower representing a system in which competition between address spaces has been eliminated by giving each address space a private memory system. This figure shows that inter-context competition has major impact on instruction cache behavior for realistic X11 workloads.

4.2.2. Self-Interference misses

We measured the impact of self-interference misses by replacing the direct-mapped caches in the simulator with two-way set associative caches of the same size. Figure 4-3 illustrates the combined effects of competition and associativity on MCPI for *splot* and *gs*. To isolate the impact of self-interference misses from that of competition misses, compare runs where all competition misses are eliminated (*bench-nocomp* and *bench-a+nocomp*). In this comparison, all misses eliminated by associativity are from conflicts within an address space. For both *gs* and *splot*, associativity eliminates a significant number of self-interference misses.

Figure 4-3 also shows the effect of associativity on inter-context competition. The two-way set associative caches eliminate some inter-context competition, but not all (compare *bench-assoc* and *bench-a+nocomp*).

4.2.3. Summary

For the two realistic X11 workloads we consider, both inter-context competition and self-interference misses have significant impact on memory system behavior, with the most significant effects occurring in the instruction cache. Strategies have been described [18] for avoiding text conflicts within an address space, but it is difficult to envision a practical software system to avoid competition between address spaces. The problem could potentially be addressed in hardware with cache associativity, although this could have an impact on machine cycle time. Many current systems rely on good luck to avoid inter-context competition. As the gap between CPU and memory speed grows, and as users demand improved performance for interactive and multi-address space systems, more aggressive cache designs may be required. The lack of client/server workloads in standard benchmark suites could lead hardware developers to believe that inter-context competition is a non-issue. Our measurements suggest it deserves more attention.

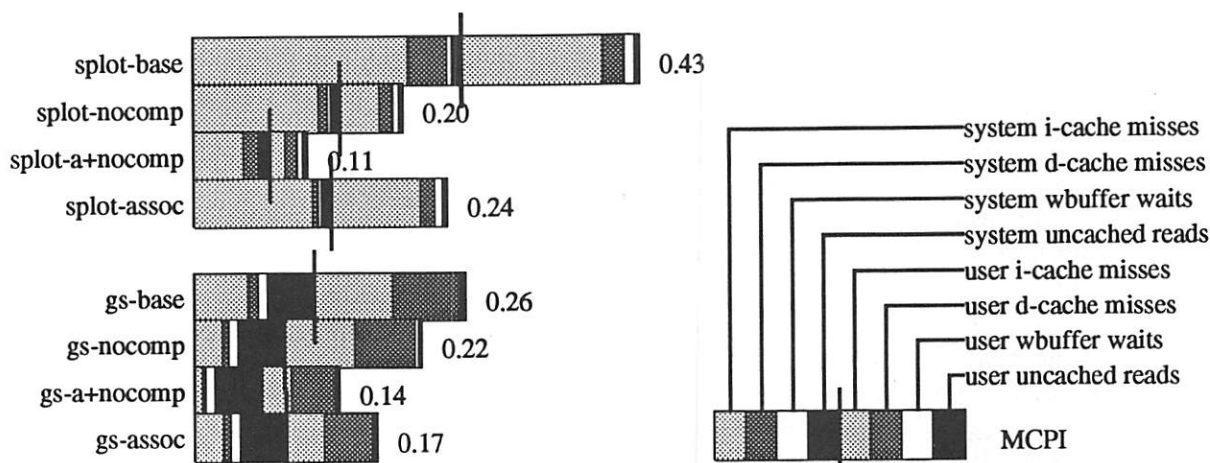


Figure 4-3: Inter-Context Competition, Associativity, and MCPI.

This figure shows the effect competition and self-interference misses on memory system performance. Each horizontal bar represents total MCPI for a given experimental run, as in Figure 4-1. We show four bars for each workload: the base system (*bench-base*), without competition (*bench-nocomp*), with associativity and without competition (*bench-a+nocomp*), and with associativity (*bench-assoc*). This figure shows that both competition and self-interference misses have significant impact on memory system behavior for realistic X11 workloads, particularly in the instruction cache.

4.3. TLB Behavior

In this section we consider the TLB behavior of the realistic X11 workloads. Table 4-1 shows TLB miss data for *splot*, *gs*, and *gcc*. Compared to other integer workloads, X11 applications have poor TLB behavior. Both *splot* and *gs* show significantly higher miss rates than *gcc*, which is relatively demanding among integer workloads [7]. Three phenomena contribute to increased TLB miss rates:

- The X11 server needs over 200 page mappings to address the entire frame buffer. Any operation that paints a significant part of the screen will tend to flush the TLB.
- The X11 server, *gs* and *splot* all have relatively large program text. This increases the likelihood that localities will be spread across multiple text pages, which in turn increases the demand on TLB resources.
- X11 applications involve two interacting user contexts, as opposed to one context for *gcc*. Multiple contexts mean more fragmentation and increased competition for limited TLB resources.

workload	user	system
splot	1.58	0.28
gs	1.49	0.14
gcc	1.11	0.07

Table 4-1: TLB Misses per 1000 instructions.

System TLB misses include misses to both user and system segments.

During the run of *splot*, 280000 user TLB misses occurred. There were about 680000 during *gs*. Estimating 20 cycles to service a TLB miss [21], the penalty for TLB faults is less than 0.04 CPI for both workloads. Thus, the impact on overall performance is not significant for the memory system we simulated.

The impact of the TLB on overall performance is dependent on the performance balance of memory system components. Processors such as the DEC Alpha 21064 [2] rely on fewer TLB entries with larger and oversized pages to achieve good TLB behavior. If software systems such as X11 don't make good use of these new features, miss rates will go up, increasing the impact of the TLB on overall performance. Also, the penalty for a single TLB miss could increase. An earlier study used 100 cycles as an estimate of the TLB miss penalty for a futuristic machine [8]. As the balance of TLB to cache resources changes, TLB performance could become an important issue. X11 workloads require more TLB resources than popular benchmarks such as *gcc*. If computer systems are designed to optimize the performance of the popular benchmarks, systems such as X11 can be expected to suffer.

Our measurements show degraded TLB behavior for X11 workloads as compared to other integer codes. Researchers at the University of Michigan [21] measured similar TLB behavior for the Mach 3.0 operating system [1, 13], another system where a user-level server contributes significantly to overall activity. The two independent studies suggest a broader conclusion, that page behavior for user-level client/server systems induces substantially elevated TLB miss rates.

As a final note, competition also affects TLB behavior. In our measurements, competition accounted for 60% of user TLB misses in *splot* and 30% of user TLB misses in *gs*. Note that additional associativity can't help here, as the DECstation 5000/200 TLB is already fully associative.

5. Conclusions

Memory system behavior for X11 workloads differs significantly from the batch-mode programs typically used in memory system studies. With large program text, a large mapped frame buffer, and multiple competing contexts, they can present a far greater load on instruction caches and TLBs than typical throughput benchmarks.

Cache associativity and TLB size are sensitive issues for hardware designers. For many machines, increasing these parameters would have direct impact on machine cycle time, the principle metric driving performance improvements in microprocessors. As machine cycle time dominates performance for many current benchmarks, there is a potential conflict between high throughput for benchmarks, and low latency for large client/server systems. Optimal throughput and optimal latency may not be possible in the same machine.

Our measurements are specific to X11 clients and server, but similar behavior can be expected in other client/server systems. Instruction cache and TLB behavior is aggravated by large user text, frequent and mandatory context switches, and multiple active contexts. Any system with these characteristic will probably have similar behavior. In systems with multiple heavy-weight servers, contention for memory system resources will be even more intense.

6. Acknowledgements

Thanks to Norm Jouppi who first suggested looking at X11. Brian Bershad and Doug Tygar made useful comments on the text. Joel McCormick answered questions about X11. David Wall provided the initial version of *epoxie*. The design of the tracing system is based on prior work by Anita Borg, and her contributions continued throughout this project. Thanks also to Digital Equipment Corporation for their generous support.

References

1. Michael J. Accetta, Robert V. Baron, William Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael W. Young. Mach: A New Kernel Foundation for Unix Development. Proceedings of the Summer 1986 USENIX Conference, July, 1986, pp. 93-113.
2. Digital Equipment Corporation. Digital's 21064 Microprocessor. Data sheet.
3. Brian N. Bershad, Richard P. Draves, and Alessandro Forin. Using Microbenchmarks to Evaluate System Performance. The Proceedings of the Third Workshop on Workstation Operating Systems, April, 1992, pp. 148-153.

4. Anita Borg, R.E. Kessler, Georgia Lazana, and David Wall. Long Address Traces from RISC Machines: Generation and Analysis. WRL Research Report 89/14, Digital Equipment Corporation Western Research Laboratory, 1989.
5. J.R. Boyton, S.L. Chakrabarti, S.P. Hiebert, J.J. Lang, J.R. Owen, K.A. Marchington, P.R. Robinson, M.H. Stroyan, J.A. Waitz. "Sharing access to display resources in the Starbase/X11 Merge". *Hewlett Packard Journal* 40, 6 (December 1989), 20-32.
6. Kenneth H. Bronstein, David J. Sweetser, and William R. Yoder. "System design for compatibility of a high-performance graphics library and the X Window System.". *Hewlett Packard Journal* 40, 6 (December 1989), 6-10. .
7. J. Bradley Chen and Brian N. Bershad. The Impact of Operating System Structure on Memory System Performance. Proceedings of the 14th ACM Symposium on Operating System Principles, December, 1993.
8. J. Bradley Chen, Anita Borg, and Norman P. Jouppi. A Simulation Based Study of TLB Performance. The Proceedings of the 19th Annual International Symposium on Computer Architecture, May, 1992, pp. 114-123.
9. H. Davis, S.R. Goldschmidt, and J. Hennessy. Tango: A Multiprocessor Simulation and Tracing System. Proceedings of the International Conference on Parallel Processing, August, 1991, pp. 99-107.
10. Jeffrey D. Gee, Mark D. Hill, Dionisios N. Pnevmatikatos, and Alan Jay Smith. Cache Performance of the SPEC Benchmark Suite. University of Wisconsin-Madison, 1991.
11. Aaron Goldberg and John Hennessy. MTOOL: A Method for Detecting Memory Bottlenecks. WRL Technical Note TN-17, Digital Equipment Corporation Western Research Laboratory, 1990.
12. Aaron J. Goldberg and John L. Hennessy. "MTOOL: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications". *IEEE Transactions on Parallel Processing* 4, 1 (January 1993), 28-40.
13. David Golub, Randall Dean, Alessandro Forin and Richard Rashid. UNIX as an Application Program. Proceedings of the Summer 1990 USENIX Conference, June, 1990, pp. 87-95.
14. John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Palo Alto, CA, 1990.
15. Margaret Martonosi, Anoop Gupta, and Thomas Anderson. MemSpy, Analyzing Memory System Bottlenecks in Programs. Proceedings of the 1992 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, June, 1992, pp. 1-12.
16. Joel McCormack. Writing Fast X Servers for Dumb Color Frame Buffers. WRL Research Report 91/1, Digital Equipment Corporation Western Research Laboratory, 1991.
17. Joel McCormack and Bob McNamara. A Smart Frame Buffer. WRL Research Report 93/1, Digital Equipment Corporation Western Research Laboratory, 1993.
18. Scott McFarling. Program Optimization for Instruction Caches. The Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, April, 1989, pp. 183-191.
19. Jeffrey C. Mogul and Anita Borg. The Effect of Context Switches on Cache Performance. The Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, April, 1991, pp. 75-84.
20. Jeffrey C. Mogul. SPECmarks are leading us astray. The Third Workshop on Workstation Operating Systems, April, 1992, pp. 160-161.
21. David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest, Trevor Mudge and Richard Brown. Design Tradeoffs for Software-Managed TLBs. Proceedings of the 20th Annual International Symposium on Computer Architecture, May, 1993, pp. 27-38.

22. J.L. Peterson. XSCOPE: a debugging and PERFORMANCE tool for X11. Proceedings of the IFIP 11th World Computer Congress, September, 1989, pp. 49-54.
23. *SPEC Benchmark Suite Release 1.0*. System Performance Evaluation Cooperative, 1989.
24. David W. Wall. Systems for Late Code Modification. In *Code Generation --- Concepts, Tools, Techniques*, Springer-Verlag, 1992, pp. 275-293.

J. Bradley Chen is presently completing a PhD in Computer Science at Carnegie Mellon University. His interests include operating systems, memory systems, and issues relating to the design and performance of large software systems. He received BS and MS degrees in 1987 from Stanford University.

A Uniform Name Service

for Spring's UNIX Environment

Michael N. Nelson* Sanjay R. Radia
Sun Microsystems, Inc.
Mountain View, CA 94043 USA

Abstract

The Spring operating system provides a uniform name service that can be used to associate any name with any object independent of the type of object, and allows arbitrary name spaces to be created and used as first-class objects. We have used this name service to unify the many UNIX[®] name spaces. Objects that on UNIX systems are typically stored in separate name spaces are all accessible via a single uniform name service in Spring. In addition, it is easy to add new Spring objects that are not currently available in UNIX systems without modifying the underlying name service.

1. Introduction

Typical UNIX systems have various kinds of nameable objects, such as files, printers, and services, and have several name services, each tailored for a specific kind of object. Such *type-specific* name services are usually built into the subsystem implementing the specific type of object. Examples of the many type-specific name services in typical UNIX systems include:

- A UNIX file system which provides its own mechanism for naming file objects, including operations for binding names to files and accessing files by name. The file system is also responsible for storing and managing the database of name-to-object bindings.
- The name space for printers which is stored in */etc/printcap* and is used by the various printing commands.
- Light weight name services such as environment variables which have their own name space with library implementations.
- Distributed versions of UNIX systems usually have one or more higher level name services called *directory* services such as Sun's NIS which are used for resource location, mail addressing, and authentication. Directory services bind names to data values.

Each of these name services has its own syntax for names, and its own interface and implementation for performing operations on the name space. The client is burdened with the requirement of dealing with different names and name services depending on what objects are to be accessed. Disjoint parts of the name space must be used for different types of objects since, for example, it is not possible to bind a file as an environment variable. Another problem is that the non-uniform name spaces generally cannot be composed together. For example, the name spaces for files cannot be attached to the name spaces for environment variables. Furthermore, defining new objects or services that are accessed by name is difficult, since the new objects must be made to "fit in" somewhere or a new name space must be defined and implemented.

* Authors current affiliation: Silicon Graphics, Inc. (mnelson@sgi.com)

In systems such as the UNIX system that do not have a unifying notion of objects, a popular technique has been to make each object provide the file interface and for each object implementation to implement the file system naming interface (the file being the most versatile and well defined entity in the system). Plan 9 [1, 2], has used this approach extensively. Although it is possible to make many objects look like files, it would be better if this constraint were not necessary since the file interface is limited in capturing the semantics of all objects. Furthermore, it is difficult to make every object behave like a file; some object types and name spaces cannot be easily fit into the file name space.

This paper describes the use of a uniform object-oriented naming system to unify many UNIX name spaces. We compare and contrast our approach with that of Plan 9; although there are similarities, there are many subtle and important differences. The naming system described in this paper was developed as part of the Spring operating system project. This naming system works together with the Spring UNIX subsystem [3] to provide users and normal UNIX programs uniform naming access to most types of UNIX objects.

2. Spring Operating System

Spring is a distributed, multi-threaded, extensible operating system that is structured around the notion of *objects*. A Spring object is an abstraction that contains state and provides a set of operations to manipulate that state. The description of the object and its operations are specified in an *interface definition language* (IDL). IDL supports both notions of *single* and *multiple* interface inheritance.

A Spring *domain* is an *address space* with a collection of *threads*. A given domain may act as the server (implementor) of some objects and the clients of other objects. The server and the client can be in the same domain or in different domains. In the latter case, the representation of the object includes an unforgeable nucleus *door identifier* that identifies the server domain [4].

The Spring kernel supports basic cross domain invocations and threads, low-level machine-dependent handling, as well as basic virtual memory support for memory mapping and physical memory management [4, 5]. A Spring kernel does not know about other Spring kernels—all remote invocations are handled by a *network proxy* server.

A typical Spring node runs several servers besides the kernel. These include a name server, file servers, a linker domain that manages and caches dynamically linked libraries, a network proxy that handles remote invocations, a device server that provides basic terminal handling as well as frame-buffer and mouse support, and a UNIX server that provides support for running UNIX binaries on Spring [3].

3. Spring Naming

The Spring name service [6] allows any object to be associated with any name. A name-to-object association is called a *name binding*. Each name binding is stored in a context. A *context* is an object that contains a set of name bindings in which each name is unique [7, 8]. An example of a context is a UNIX file directory. An object can be bound to several different names in possibly several different contexts at the same time.

Resolving a name is an operation on a context to obtain the object denoted by the name; binding a name is an operation to associate a name with a particular object. These operations return or take as a parameter the object itself, not a lower-level identifier for the object as some systems do [9, 10].

Since a context is like any other object, it can also be bound to a name in some context. By binding contexts we can create a *naming graph*. The UNIX file system is a naming graph that is frequently restricted to a tree. Given a context in some naming graph, a sequence of names can be used to refer to an object relative to that context. Such a sequence of names is called a *compound name*. A name with a single component is called a *simple name*. Informally, we refer to the naming graph spanned by a context as a name space, which includes all bindings of names and objects that are accessible directly or indirectly through that context.

Spring contexts provide support for the Spring security model. When an object is bound, an ACL can be given that specifies which principals are allowed which rights for the object. When an object is resolved, a set of desired *modes* is specified. Modes are a superset of rights. For example, read and write modes correspond directly to read and write access rights; however, append mode implies write access but also indicates the “mode” with which the object should be accessed when writes occur. When a mode is specified to a resolve operation, an object with the desired modes is returned if the client doing the resolve is allowed the corresponding rights.

Contexts and name spaces are first class entities in Spring: a context, and the name space it spans, can be manipulated directly. A context can be passed around like any other object. For example, two applications can exchange and share a private name space. In the UNIX system, such applications would have had to build their own naming facility or incorporate the private name space into a larger system wide name space and access it indirectly via the root or working context. Name spaces can be composed by binding a context to a name in some name space (this is a generalization of the mount operation). The first class nature of contexts also makes it very simple to support ordered merges as described below.

3.1. Names

A name consists of a sequence of one or more components. Each component is an unordered set of elements. The identifier element of a component is an arbitrary UNICODE string that is never parsed (only compared for equality) by the name service. Other elements of a component encode version numbers (allowing references to the latest version) and an object “kind” (analogous to file name extensions). The presentation and parsing of names is relegated to user interface software. The current syntax that is used for Spring names is identical to the syntax used for UNIX path names.

3.2. Naming Operations

The primary interface between a client and the name service (the context interface) is simple. The following are the common operations on contexts:

- `named_object = context.resolve (name, mode)`
Returns the object denoted by the name. The mode argument indicates intended use of the object.
- `context.bind (name, binding_type, named_object, acl)`
Establishes a binding in the context (or in a context reachable from it, if the name is a compound name). The binding type is used to distinguish bindings that the name service has to process during resolution. Symbolic links specify *symbolic_binding*, name spaces are grafted by specifying *context_binding*, and normal objects specify *normal_binding*. The *acl* is the binding’s access control list.
- `new_context = context.bind_new_context(name, acl)`
Creates a new context with a particular name. It is also possible to create contexts that are (as yet) unnamed, although that is an operation on the name server interface, not the context interface.
- `iterator = context.get_all_bindings(name)`
Returns the binding information in the context.
- `context.unbind(name)`
Deletes a binding

3.3. Ordered Merges

Given a set of contexts one can construct a new context using that set. An ordered merge context in Spring (like union mounts in Plan 9) is a context whose implementation contains several contexts. The result is that the name spaces of all of the involved contexts are merged. We use ordered merges for constructing the per-domain name space in Spring and also for search paths.

Ordered merges in other systems are usually restricted to special situations such as command search paths or unions at mount time (as in Plan 9). In Spring, since contexts objects can be manipulated directly, we did not clutter the context interface with a notion of merges. Instead, we provided it as a separate context implemented in a library. We give several examples below of our use of ordered merges in constructing name spaces.

3.4. The Spring Naming Environment

The Spring name service imposes no policies such as the notion of a global root. Such notions are built as policies on the top of the name service. The policies that we use in Spring are based on a private per-domain (per-process) view of naming coupled with shared name spaces. Below we first describe the shared machine and village name spaces; these name spaces are used to construct part of the per-domain private name spaces. Our per-domain name spaces are based on many of the ideas in [11] and are similar to the per-process name spaces in Plan 9.

3.4.1. Machine and Village Name Spaces

Each machine has a per-machine name space. This name space is implemented by a per-machine name server and contains the domain name space, all machine-specific services, and the devices exported from the machine.

A collection of machines is called a village. Each village has a per-village name space. This name space is implemented by one of the machines in the village and contains information global to the village such as:

- the name spaces for all of the machines. Each machine name server binds a context that represents the machine's name space into the village.
- globally accessible services such as NIS.

3.4.2 Per-Domain Name Space

Each domain has its own private name space that can be customized to access arbitrary name spaces such as the machine and village name spaces. A per-domain name space gives great flexibility in sharing objects and name spaces in a distributed environment. It has also allowed us to incorporate private name spaces such as the environment variables name space. Ordered merge constructions are used for further flexibility in tailoring name spaces. The result is a powerful naming environment. A domain's private name space typically contains (see Figure 1):

- Private name bindings

The domain's name space has bindings for environment variables and program input/output objects. (A special mechanism to pass standard IO objects as implicit parameters is not needed.)

- Shared name spaces

Several shared name spaces are attached to the domain's name space using well-known names: *user* (the name spaces of different users), *~* (the home name space of the user owning the domain), and *dev* (devices). If there were, for example, a worldwide global name space, we would attach it to the name space of a domain using a well-known name.

- Generic name spaces containing standard system objects

A domain's name space has generic name spaces that contain system objects: *sys* (executables under *sys/bin* and libraries under *sys/lib*) and *services* (such as *services/authentication*). The name spaces of *sys* and *services* have parallel structures in both the machine and village. These name spaces contain copies of such system objects for local use¹.

The *sys* and *services* name spaces in a domain's private name space are typically an ordered merge of the corresponding name spaces of the machine and village. The merge arranges for the local instance to be visible first. A user can also keep similar structures in his home contexts which can be used for further tailoring of these name spaces. During remote execution, the merges are automatically reevaluated to take advantage of the name spaces of the remote site.

1. The local copies are in addition to what is kept automatically by our caching scheme. Local copies allow a machine to maintain local autonomy, especially during disconnected operation.

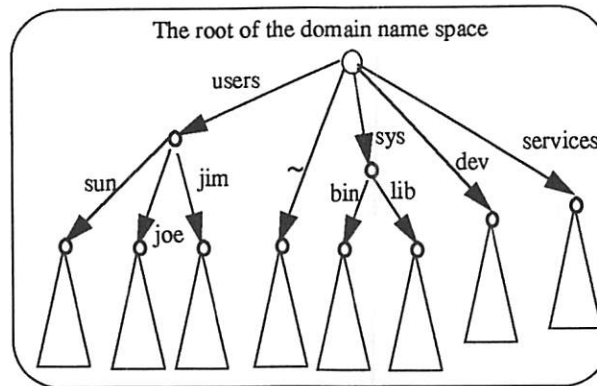


Figure 1. A Typical Per-Domain Name Space

Each parent domain passes to its child a context defining the child's private name spaces. This is usually a copy of the parent's context. Since most domains make few changes to their name spaces, we can make the child's context be a copy-on-write copy of the parent's. The init domain hand-crafts the name space given to its children. When a user logs in, the name space is further modified according to the user's profile.

4. Implementing UNIX Name Spaces

We have used the Spring naming system to unify many traditional UNIX name spaces. In this section we will describe our implementation of the various UNIX name spaces.

4.1. The UNIX Name Space on Spring

In order to allow UNIX applications to run on Spring we provide the UNIX naming environment via the per-domain name space. The UNIX root becomes the per-domain context of the domain that implements the UNIX process. The current working directory becomes a symbolic link called ".cwd". We add bindings for /bin, /lib, /etc, /usr, and other generic UNIX contexts. The per-domain context often contains additional Spring entries such as Spring commands and libraries to allow UNIX users on Spring to access Spring resources. This is accomplished by an ordered merge; for example /bin is an ordered merge of the context containing UNIX commands and the context containing Spring commands.

4.2. Generic Naming Implementation

The Spring naming system is comprised of several name servers. Most of these name servers run generic name server code that can store objects of any type. This code is available in a library and can be used to build name servers on native Spring and under Spring emulation on the SunOS system. The only name spaces that are not implemented by the generic name server library code are the file system and NIS.

4.3. File System

The Spring file system supports file servers that have an integrated name service such as that in the UNIX system. In addition we have extended typical UNIX file system naming in two ways:

- Files can be bound in other non-file-system parts of Spring name spaces.
- Non-file Spring objects can be bound into file system name spaces.

We also allow access to file systems on UNIX machines on the network via gateway name servers that translate names between Spring and the UNIX system.

The UNIX file system name space maps directly into the Spring name service. UNIX files are mapped to Spring files and UNIX directories are mapped to Spring contexts.

Files are opened by resolving them using a particular context. For example, in order to open the file “/etc/passwd” read-write relative to the per-domain context, the following call would be issued:

```
file f = context.resolve("etc/passwd", access_read_write);
```

4.4. NIS

The NIS name space is hierarchical in nature and as such can be mapped into the Spring name service. The NIS name server exports a context that contains the default NIS domain. The context for each domain contains a context for each map. The context for each map contains all of the keys. A resolution of a key returns a string that contains the value. For example to lookup mnn's passwd entry the following call would be issued on a context that represented the root of the NIS name space:

```
string s = context.resolve("Wildfire/passwd/mnn", access_read);
```

4.5. Environment variables

Environment variables are bound into the per-domain context. For example, looking up the value of the HOME environment variable would require the following call:

```
string s = context.resolve("HOME");
```

4.6. Domains

On SVR4 systems, information about processes can be acquired via the /proc file system. Information can be found in the various files that are stored in the directory for a given process. In order to perform an operation on the process, the read, write, and ioctl interface must be used on the appropriate file.

In Spring we export information about domains via a domain object. The domain object of a domain and other objects that the domain wants published to the outside world are bound in the per-machine name space of the machine on which the domain executes. These objects can be manipulated directly by invoking operations rather than encoding the request into an ioctl or an ascii string to a write call. For example to stop all of the threads in domain 27 one would issue the following calls:

```
domain d = context.resolve("domains/27", access_read);  
d->stop();
```

4.7. Devices

Device objects are accessed via the dev context as in the UNIX system. The dev context is a merge of the per-domain dev context (which contains stdio objects), the dev context on the machine (which contains devices on that machine), and the dev context of the village. The dev context of the village can be used to provide special devices available on particular machines that are meant to be made widely available. In addition one can also access a device on another machine by using a name of the form “village/machine/machine_name/dev/device_name”.

4.8. UNIX I/O Uniformity

One of the advantages of the Plan 9 approach of making everything look like a file is that it provides a uniform object interface to clients of the name service. In Plan 9, all objects are files so all objects can be accessed via read and write calls. Unfortunately, it is unknown what is appropriate to pass to a write call or expect from a read call. The semantics of the read or write call depend on the type of the object involved.

In Spring we still provide a common I/O interface for those objects for which I/O is appropriate. This is done by having each object that supports the I/O interface inherit from the *io* class. If a program gets an object that it wishes to treat as an *io* object, then it can do a type system operation to traverse the type hierarchy to the *io* class. If this operation succeeds, then the object can be treated as an *io* object meaning that read and write operations can be invoked on the object.

In Spring the semantics of read and write operations are well defined. We do not attempt to overload the read and write operations to provide functionality that is best provided by methods on the object itself.

5. Related Work

Other object oriented systems have either not provided a uniform name service or provided one with serious restrictions. Programming language based object oriented systems such as Smalltalk rely on the name space of variables provided by the programming language. Others like Choices [10] have the serious restriction that the object managers must be integrated and reside with the name service; this makes it difficult to add new object types. Furthermore, Choices provides two name services: one for persistent objects and one for transient objects.

DCE [12] has made an attempt to provide uniformity by composing name spaces such as the file system name space onto the higher level directory name space. This is done through special entities called junctions. For example a junction point in the directory level name space called *fs*, signals that one is entering the file system name space (it is like a specialized mount point). What this provides is network-wide access to files: there is a uniform way of naming files and file systems in other parts of the network. The client is still faced with multiple name spaces and their corresponding naming interfaces. Furthermore, it is not clear how easy it is to create new junctions when introducing new object types and their corresponding name spaces. Thus, DCE does not provide a uniform name service but instead provides a uniform way of allowing network-wide access to the various name spaces.

Providing a uniform name service has been difficult in the UNIX system because the entities being named have different representations. Type-specific name services provide tokens that must be used for a particular purpose, since the type is assumed; for example resolving a file name as part of the open operation returns an integer which is used for file operations. On the other hand, directory services provide values that must be resolved at another level to be useful; for example, looking up a host returns a byte string that is a network address.

The Plan 9 system has tried to provide uniform naming in a manner different than we have used. Plan 9 unifies the UNIX name spaces by making each nameable object provide the file interface and having each object implementation implement the file system naming interface. Given the lack of some unifying concept like an object, the Plan 9 approach is probably the best way of providing uniform naming in the UNIX system, or other non-object oriented systems. The main advantage of the Plan 9 naming model is that it fits directly into the UNIX way of doing things: UNIX users and standard UNIX tools already understand files and file systems so it is relatively easy for users and standard UNIX tools to use the Plan 9 system. However, with the trend towards object oriented systems, we believe that our approach is better than Plan 9's approach of attempting to make all objects look like files. In the rest of this section we will discuss why we believe that our object-oriented naming system is better than the Plan 9 file centric naming view. We will also discuss Plan 9's more flexible notion of a per-process mount table.

There are basically four major advantages of the Spring naming approach over the Plan 9 approach. The first advantage comes from Spring's object-oriented approach. In Spring, nameable objects have strong well-defined interfaces that are appropriate for the particular type of object. We do not attempt to make all objects obey the file interface and hide the object's true interface inside the *read*, *write*, and *ioctl* operations.

The second advantage of the Spring approach is that the effort required to make new object types nameable is low. In Spring, nameable objects do not have to made to obey the file interface and object managers do not have to implement the name service. For example, objects implemented by the Spring kernel such as domains (processes), VM objects, and kernel monitoring objects are routinely bound into the machine's name spaces; the kernel did not have to change the nature of these objects or implement a name service to make this possible. This means that we can

easily add new nameable objects and new implementations of nameable objects to the system at any time. If someone comes up with a new object type, they do not have to modify the implementation of the name service to allow this object to be stored in a name space. Spring developers routinely create new objects and make them available via the name service.

In contrast to Spring, the Plan 9 approach requires that nameable non-file objects be made to look like files and implementations of the non-file objects must implement the file system directory operations. Although Plan 9 has been able to make many objects look like files, it does require a certain amount of effort which is not required in our system. Furthermore, we believe that the Plan 9 approach cannot succeed for all kinds of object. Indeed the network name space in Plan 9 (the name space that binds file servers) does not map very easily to a file system and hence is a separate non-file name service.

The third advantage of the Spring approach is that name spaces do not need to be segregated by the type of objects being bound in them. For example, we can store a file in our equivalent of the /proc file system and we can store a domain object in the file system. This functionality is not possible in Plan 9; the best one can do in Plan 9 is to use the union mount to attach special file systems behind the regular directory so as to allow regular files to be created "ahead" of the mounted special file system. Thus in spite of making most objects look like files, a lack of uniformity is evident since each file server can only bind objects of the same true type.

The last advantage of the Spring approach is the notion that contexts and name spaces are first class entities. In Spring, contexts can be manipulated directly and passed around like any other object. In the UNIX system, applications that need to exchange and share a private name space would have to build their own naming facility or incorporate the private name space into a larger system wide name space and access it indirectly via the root or working context.

The first class nature of contexts has made it very simple for us to support context constructs like ordered merges. A client can construct an ordered merge context from a set of contexts directly; the resulting context can be used like any other context: it can be passed around, bound to a name, etc. We did not have to resort to special name resolution rules or mechanisms. If we were not able to manipulate contexts directly, we might have been forced to limit the use of ordered merges to "mount" time (as in Plan 9) and to make the notion of merges be a part of the naming interface. We also allow ordered merges to be specified as symbolic links in which case the meaning of the merge is evaluated when a name is resolved through that name binding. We believe such symbolic ordered merges could easily be added to Plan 9.

One place where Plan 9 is more flexible than Spring is Plan 9's per-process mount table. In Spring mounting a naming tree on a shared sub-tree makes the mounted tree visible to all processes that share the sub-tree. In Plan 9 a naming tree would only be visible to the process doing the mount (this fits naturally with the UNIX system notion of a mount table). Plan 9's per-process mount table is clearly more flexible. We decided not to pursue the per-process mount table for two reasons. First, we found that ordered merges allow a process sufficient flexibility in configuring its name space and have not found the need to add the notion of per-process mount tables. Second, when a private name space overlays a shared name space, resolving names in the shared name space becomes more complex in a distributed environment: one cannot blindly resolve pathnames in a remote shared name space as private local mounts may overlay parts of it.

6 Status

The Spring name service described in this paper has been implemented as part of the Spring operating system which is currently running on SPARCstation 1, 2, and 10 machines. A general name server implementation that is used for the machine, village, and per-domain name spaces is available as a library for Spring programs. This library is usable both by programs running on native Spring and programs running under a Spring emulation package on SunOS.

7 Conclusion

The Spring name service makes it possible to unify the many UNIX name spaces. Each of these name spaces fits seamlessly into the overall Spring name service. Our success at unifying the name spaces is evident from the fact that standard UNIX tools such as *ls* and *filemanager* can be used to browse any Spring name space without changing the tools. Thus users now can easily browse name spaces that once required special tools.

Spring provides more than just a simple unification of UNIX name spaces. Our use of object-orientation means that users not only see a uniform name space but they also get the advantages of strong interfaces and the ability to add new nameable objects and new implementations at any time without modifying the name service. The Spring name service is more powerful in allowing contexts and name spaces to be manipulated as first class entities. We believe that a powerful and uniform object-oriented name service like that used in Spring will provide the right infrastructure to allow the complex and diverse systems of the future to be usable by the average user.

8 References

- [1] Pike R., Presotto D., Thompson, K., and Trickey, H., "Use of Name Spaces in Plan 9", Unpublished report, 1992.
- [2] Pike R., Presotto D., Thompson, K., and Trickey, H., "Plan 9 from Bell Labs", *Proceedings of 1990 UKUUG Conference*, July, 1990, pp. 1-9.
- [3] Khalidi, Y. A. and Nelson, M. N., "An Implementation of UNIX on an Object-oriented Operating System", *Proceedings of the 1993 Winter USENIX Conference*, January 1993, pp. 469-479.
- [4] Hamilton, K.G. and Kougiouris, P., "The Spring Nucleus: A Microkernel for Objects," *Proceedings of the 1993 Summer USENIX Conference*, June 1993, pp. 147-160.
- [5] Khalidi, Y.A. and Nelson, M.N., "The Spring Virtual Memory System," Sun Microsystems Laboratories, Technical Report SMLI-92-388, September 1992.
- [6] Radia, S. R., Nelson, M. N., and Powell, M. L., "The Spring Name Service," Sun Microsystems Laboratories, Technical Report SMLI-93-16, October 1993.
- [7] Saltzer, J. H., "Naming and Binding Objects," in *Operating Systems: An Advanced Course*, volume 60, pages 99--208. Springer-Verlag, New York, 1978.
- [8] Radia, S., "Names, Contexts, and Closure Mechanisms in Distributed Computing Environments," Ph.D. thesis, Univ. of Waterloo, Dept. of Comp. Sci., Waterloo, Ontario, Canada, 1989. Also report UW/ICR 90-01.
- [9] Berabeu-Auban, J. M., et al., "The Architecture of Ra: A Kernel for Clouds", *Proceedings of the 22th Hawaii International Conference on System Sciences*, January 1989, pp. 936-945.
- [10] Campbell, R.H., Islam, N., and Madany, P., "Choices, Frameworks, and Refinement," *USENIX Computing Systems*, 5, 3 (Summer 1992), pp. 217-257
- [11] Radia, S. and Pacht, J., "The Per-Process View of Naming and Remote Execution", *IEEE Parallel and Distributed Technology*, Vol 1, No 3, August 1993, pp 71-80.
- [12] Rosenberry W., Kenny D., Fisher G., "Understanding DCE", O'Reilly & Associates, Inc., California 1993.

UNIX is a registered trademark of UNIX System Laboratories

Michael N. Nelson is currently a Member of the Technical Staff at Silicon Graphics. Before joining SGI he was one of the principal developers of the Spring Operating System at Sun Microsystems and was also a principal developer of the Sprite Operating System at UC Berkeley. His interests include distributed, object-oriented software, operating systems, and architecture. He has a Ph.D. in Computer Science from UC Berkeley. He can be reached at Silicon Graphics, 2011 N. Shoreline Blvd., Mountain View, CA 94043, USA, or via e-mail at mnelson@sgi.com.

Sanjay Radia is a senior staff engineer at Sun Microsystems, where is one of the principal developers of the Spring distributed operating system. His interests includes design and implementation of distributed software and operating systems, particularly using object-oriented technology. Previously, he has worked on several distributed system projects at I.N.I.R.A., France, and at the University of Waterloo. He has a Phd from University of Waterloo, Canada. He can be reached via email at srradia@sun.com.

ACID: A Debugger Built From A Language

Phil Winterbottom
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

ACID is an unusual source-level symbolic debugger for Plan 9. It is implemented as a language interpreter with specialized primitives that provide debugger support. Programs written in the language manipulate one or more target processes; variables in the language represent the symbols, state, and resources of those processes. This structure allows complex interaction between the debugger and the target program and provides a convenient method of parameterizing differences between machine architectures. Although some effort is required to learn the debugging language, the richness and flexibility of the debugging environment encourages new ways of reasoning about the way programs run and the conditions under which they fail.

1. Introduction

The size and complexity of programs have increased in proportion to processor speed and memory but the interface between debugger and programmer has changed little. Graphical user interfaces have eased some of the tedious aspects of the interaction. A graphical interface is a convenient means for navigating through source and data structures but provides little benefit for process control. The introduction of a new concurrent language, ALEF [Win93], emphasized the inadequacies of the existing Plan 9 [Pike90] debugger *db*, a distant relative of *adb*, and made it clear that a new debugger was required.

Current debuggers like *dbx*, *sdb*, and *gdb* are limited to answering only the questions their authors envisage. As a result, they supply a plethora of specialized commands, each attempting to anticipate a specific question a user may ask. When a debugging situation arises that is beyond the scope of the command set, the tool is useless. Further, it is often tedious or impossible to reproduce an anomalous state of the program, especially when the state is embedded in the program's data structures.

ACID applies some ideas found in CAD software used for hardware test and simulation. It is based on the notion that the state and resources of a program are best represented and manipulated by a language. The state and resources, such as memory, registers, variables, type information and source code are represented by variables in the language. Expressions provide a computation mechanism and control statements allow repetitive or selective interpretation based on the result of expression evaluation. The heart of the ACID debugger is an interpreter for a small typeless language whose operators mirror the operations of C and ALEF, which in turn correspond well to the basic operations of the machine. The interpreter itself knows nothing of the underlying hardware; it deals with the program state and resources in the abstract. Fundamental routines to control processes, read files, and interface to the system are implemented as builtin functions available to the interpreter. The actual debugger functionality is coded in ACID; commands are implemented as ACID functions.

This language-based approach has several advantages. Most importantly, programs written in ACID, including most of the debugger itself, are inherently portable. Furthermore ACID avoids the limitations other debuggers impose when debugging parallel programs. Instead of embedding a fixed process model in the debugger, ACID allows the programmer to adapt the debugger to handle an arbitrary process partitioning or program structure. The ability to dynamically interact with an executing process provides clear advantages over debuggers constrained to

probe a static image. Finally, the ACID language is a powerful vehicle for expressing assertions about logic, process state, and the contents of data structures. When combined with dynamic interaction it allows a limited form of automated program verification without requiring modification or recompilation of the source code. The language is also an excellent vehicle for preserving a test suite for later regression testing.

The debugger may be customized by its users; standard functions may be modified or extended to suit a particular application or preference. For example, the kernel developers in our group require a command set supporting assembler-level debugging while the application programmers prefer source-level functionality. Although the default library is biased toward assembler-level debugging, it is easily modified to provide a convenient source-level interface. The debugger itself does not change; the user combines primitives and existing ACID functions in different ways to implement the desired interface.

2. Related Work

DUEL [Gol93], an extension to *gdb* [Stal91], proposes using a high level expression evaluator to solve some of these problems. The evaluator provides iterators to loop over data structures and conditionals to control evaluation of expressions. The author shows that complex state queries can be formulated by combining concise expressions but this only addresses part of the problem. A program is a dynamic entity; questions asked when the program is in a static state are meaningful only after the program has been 'caught' in that state. The framework for manipulating the program is still as primitive as the underlying debugger. While DUEL provides a means to probe data structures it entirely neglects the most beneficial aspect of debugging languages: the ability to control processes. ACID is structured around a thread of control that passes between the interpreter and the target program.

The NeD debugger [May92] is a set of extensions to TCL [Ous90] that provide debugging primitives. The resulting language, NeDtcl, is used to implement a portable interface between a conventional debugger, *pdb* [May90], and a server that executes NeDtcl programs that operate on the target program. Execution of the NeDtcl programs implements the debugging primitives that *pdb* expects. NeD is targeted at multi-process debugging across a network, and proves the flexibility of a language as a means of communication between debugging tools. Whereas NeD provides an interface between a conventional debugger and the process it debugs, ACID is the debugger itself. While NeD has some of the ideas found in ACID it is targeted toward a different purpose. ACID seeks to integrate the manipulation of a programs resources into the debugger while NeD provides a flexible interconnect between components of the debugging environment. The choice of TCL is appropriate for its use in NeD but is not suitable for ACID. ACID relies on the coupling of the type system with expression evaluation, which are the root of its design, to provide the debugging primitives.

Dalek [Ols90] is an event based language extension to *gdb*. State transitions in the target program cause events to be queued for processing by the debugging language.

ACID has many of the advantages of same process or *local agent* debuggers, like Parasight [Aral], without the need for dynamic linking or shared memory. ACID improves on the ideas of these other systems by completely integrating all aspects of the debugging process into the language environment. Of particular importance is the relationship between ACID variables, program symbols, source code, registers and type information. This integration is made possible by the design of the ACID language.

Interpreted languages such as Lisp and Smalltalk are able to provide richer debugging environments through more complete information than their compiled counterparts. ACID is a means to gather and represent similar information about compiled programs through cooperation with the compilation tools and library implementers.

3. ACID the Language

ACID is a small interpreted language targeted to its debugging task. It focuses on representing program state and addressing data rather than expressing complex computations. Program state is *addressable* from an ACID program. In addition to parsing and executing expressions and providing an architecture-independent interface to the target process, the interpreter supplies a mark-and-scan garbage collector to manage storage.

Every ACID session begins with the loading of the ACID libraries. These libraries contain functions, written in ACID, that provide a standard debugging environment including breakpoint management, stepping by instruction or statement, stack tracing, and access to variables, memory, and registers. The library contains 600 lines of ACID code and provides functionality similar to *dbx*. Following the loading of the system library, ACID loads user-

specified libraries; this load sequence allows the user to augment or override the standard commands to customize the debugging environment. When all libraries are loaded, ACID issues an interactive prompt and begins evaluating expressions entered by the user. The ACID 'commands' are actually invocations of builtin primitives or previously defined ACID functions. ACID evaluates each expression as it is entered and prints the result.

4. Types and Variables

Acid variables are of four basic types: *integer*, *string*, *float*, and *list*. The type of a variable is inferred by the type of the right-hand side of an assignment expression. Many of the operators can be applied to more than one type; for these operators the action of the operator is determined by the type of its operands. For example, the + operator adds *integer* and *float* operands, and concatenates *string* and *list* operands. Lists are the only complex type in ACID; there are no arrays, structures or pointers. Operators provide head, tail, append and delete operations. Lists can also be indexed like arrays.

ACID has two levels of scope: global and local. Function parameters and variables declared in a function body using the `local` keyword are created at entry to the function and exist for the lifetime of a function. Global variables are created by assignment and need not be declared. All variables and functions in the program being debugged are entered in the ACID symbol table as global variables during ACID initialization. Conflicting variable names are resolved by prefixing enough '\$' characters to make them unique. Syntactically, ACID variables and target program symbols are referenced identically. However, the variables are managed differently in the ACID symbol table and the user must be aware of this distinction. The value of an ACID variable is stored in the symbol table; a reference returns the value. The symbol table entry for a variable or function in the target program contains the address of that symbol in the image of the program. Thus, the value of a program variable is accessed by indirect reference through the ACID variable that has the same name; the value of an ACID variable is the address of the corresponding program variable.

5. Control Flow

The `while` and `loop` statements implement looping. The former is similar to the same statement in C. The latter evaluates starting and ending expressions yielding integers and iterates while an incrementing loop index is within the bounds of those expressions.

```
acid: i = 0; loop 1,5 do print(i++)
0x00000000
0x00000004
0x00000008
0x0000000c
0x00000010
acid:
```

The traditional `if-- then-- else` statement implements conditional execution.

6. Addressing

Two indirection operators allow ACID to access values in the program being debugged. The * operator fetches a value from the memory image of an executing process; the @ operator fetches a value from the text file of the process. When either operator appears on the left side of an assignment, the value is written rather than read.

The indirection operator must know the size of the object referenced by a variable. The Plan 9 compilers neglect to include this information in the program symbol table, so ACID cannot derive this information implicitly. Instead ACID variables have formats. The format is a code letter specifying the printing style and the effect of some of the operators on that variable. The indirection operators look at the format code to determine the number of bytes to read or write. The format codes are derived from the format letters used by *db*. By default, symbol table variables and numeric constants are assigned the format code 'X' which specifies 32-bit hexadecimal. Printing such a variable yields output of the form 0x00123456. An indirect reference through the variable fetches 32 bits of data at the address indicated by the variable. Other formats specify various data types, for example *i* an instruction, *D* a signed 32 bit decimal, *s* a null-terminated string. The `fmt` function allows the user to change the format code of a variable to control the printing format and operator side effects. This function evaluates the expression supplied as the first argument, attaches the format code supplied as the second argument to the result and returns that value. If

the result is assigned to a variable, the new format code applies to that variable. For convenience, ACID provides the \ operator as a shorthand infix form of fmt. For example:

```
acid: x=10
acid: x                                     // print x in hex
0x0000000a
acid: x = fmt(x, 'D')                       // make x type decimal
acid: print(x, fmt(x, 'X'), x\X)           // print x in decimal & hex
10 0x0000000a 0x0000000a
acid: x                                     // print x in decimal
10
acid: x\o                                   // print x in octal
000000000012
```

The ++ and -- operators increment or decrement a variable by an amount determined by its format code. Some formats imply a non-fixed size. For example, the i format code disassembles an instruction into a string. On a 68020, which has variable length instructions:

```
acid: p=main\i                             // p = addr(main), type INST
acid: loop 1,5 do print(p\X, @p++)         // disassemble 5 instructions
0x0000222e LEA 0xffffe948(A7),A7
0x00002232 MOVL s+0x4(A7),A2
0x00002236 PEA 0x2f($0)
0x0000223a MOVL A2,-(A7)
0x0000223c BSR utfrrune
acid:
```

Here, main is the address of the function of the same name in the program under test. The loop retrieves the five instructions beginning at that address and then prints the address and the assembly language representation of each. Notice that the stride of the increment operator varies with the size of the instruction: the MOVL at 0x0000223a is a two byte instruction while all others are four bytes long.

Registers are treated as normal program variables referenced by their symbolic assembler language names. When a process stops, the register set is saved by the kernel at a known virtual address in the process memory map. The ACID variables associated with the registers point to the saved values and the * indirection operator can then be used to read and write the register set. Since the registers are accessed via ACID variables they may be used in arbitrary expressions.

```
acid: PC                                     // addr of saved PC
0xc0000f60
acid: *PC
0x0000623c                                 // contents of PC
acid: *PC\main
acid: *R1=10                               // modify R1
acid: asm(*PC+4)                           // disassemble @ PC+4
main+0x4 0x00006240      MOVW    R31,0x0(R29)
main+0x8 0x00006244      MOVW    $setR30(SB),R30
main+0x10 0x0000624c     MOVW    R1,_clock(SB)
```

Here, the saved PC is stored at address 0xc0000f60; its current content is 0x0000623c. The a format code converts this value to a string specifying the address as an offset beyond the nearest symbol. After setting the value of register 1, the example uses the asm command to disassemble a short section of code beginning at four bytes beyond the current value of the PC.

7. Process Interface

A program executing under ACID is monitored through the *proc* file system interface provided by Plan 9. Textual messages written to the *ctl* file control the execution of the process. For example writing *waitstop* to the control file causes the write to block until the target process enters the kernel and is stopped. When the process is stopped the write completes. The *startstop* message starts the target process and then does a *waitstop* action. Synchronization between the debugger and the target process is determined by the actions of the various messages.

Some operate asynchronously to the target process and always complete immediately, others block until the action completes. The asynchronous messages allow ACID to control several processes simultaneously.

The interpreter has builtin functions named after each of the control messages. The functions take a process id as argument. Any time a control message causes the program to execute instructions the interpreter performs two actions when the control operation has completed. The ACID variables pointing at the register set are fixed up to point at the saved registers, and then the user defined function `stopped` is executed. The `stopped` function may print the current address, line of source or instruction and return to interactive mode. Alternatively it may traverse a complex data structure, gather statistics and then set the program running again.

Several ACID variables are maintained by the debugger rather than the programmer. These variables allow generic ACID code to deal with the current process, architecture specifics or the symbol table. The variable `pid` is the process id of the current process ACID is debugging. The variable `symbols` contains a list of lists where each sublist contains the symbol name, its type and the value of the symbol. The variable `registers` contains a list of the machine-specific register names. Global symbols can be referenced directly by name from an ACID program. Local variables are referenced using the colon operator as `function:variable`

8. Source Level Debugging

ACID provides several builtin functions to manipulate source code. The `file` function reads a text file, inserting each line into a list. The `pcfile` and `pcline` functions each take an address as an argument. The first returns a string containing the name of the source file and the second returns an integer containing the line number of the source line containing the instruction at the address.

```
acid: pcfile(main)                // file containing main
main.c
acid: pcline(main)                // line # of main in source file
11
acid: file(pcfile(main))[pcline(main)] // print that line
main(int argc, char *argv[])
acid: src(*PC)                    // print statements near that line
9
10 void
>11 main(int argc, char *argv[])
12 {
13     int a;
```

In this example, the three primitives are combined in an expression to print a line of source code associated with an address. The `src` function prints a few lines of source around the address supplied as its argument. A companion routine, `Bsrc`, communicates with the external editor `sam`. Given an address, it loads the corresponding source file into the editor and highlights the line containing the address. This simple interface is easily extended to more complex functions. For example, the `step` function can select the current file and line in the editor each time the target program stops, giving the user a visual trace of the execution path of the program. A more complete interface allowing two way communication between ACID and the `acme` user interface [Pike93] is under construction. A filter between the debugger and the user interface provides interpretation of results from both sides of the interface. This allows the programming environment to interact with the debugger and vice-versa, a capability missing from the `sam` interface. The `src` and `Bsrc` functions are both written in ACID code using the `file` and `line` primitives. ACID provides library functions to step through source level statements and functions. Furthermore, addresses in ACID expressions can be specified by source file and line. Source code is manipulated in the ACID *list* data type.

9. The ACID Library

The following examples define some useful commands and illustrate the interaction of the debugger and the interpreter.

```

defn bpset(addr)                                // set a breakpoint
{
    if match(addr, bplist) >= 0 then
        print("breakpoint already set at ", fmt(addr, 'a'), "\n");
    else {
        *fmt(addr, bpfmt) = bpinst;           // plant the breakpoint
        bplist = append bplist, addr;         // add to list
    }
}

```

The `bpset` function plants a break point in memory. The function starts by using the `match` builtin to search the breakpoint list to determine if a breakpoint is already set at the address. The indirection operator, controlled by the format code returned by the `fmt` primitive, is used to plant the breakpoint in memory. The variables `bpfmt` and `bpinst` are ACID global variables containing the format code specifying the size of the breakpoint instruction and the breakpoint instruction itself. These variables are set by architecture-dependent library code when the debugger first attaches to the executing image. Finally the address of the breakpoint is appended to the breakpoint list, `bplist`.

```

defn step()                                     // single step
{
    local lst, lpl, addr, bput;

    bput = 0;                                  // sitting on a breakpoint
    if match(*PC, bplist) >= 0 then {
        bput = fmt(*PC, bpfmt);               // save current address
        *bput = @bput;                        // replace it
    }

    lst = follow(*PC);                         // get follow set

    lpl = lst;
    while lpl do {                             // place break points
        *(head lpl) = bpinst;
        lpl = tail lpl;
    }

    startstop(pid);                            // do the step

    while lst do {                             // remove the breakpoints
        addr = fmt(head lst, bpfmt);
        *addr = @addr;                       // replace it
        lst = tail lst;
    }
    if bput != 0 then
        *bput = bpinst;                      // replace saved breakpoint
}

```

The `step` function executes a single assembler instruction. If the PC is sitting on a breakpoint, the address and size of the breakpoint are saved. The breakpoint instruction is then removed using the `@` operator to fetch `bpfmt` bytes from the text file and to place it into the memory of the executing process using the `*` operator. The `follow` function is an ACID builtin which returns a follow-set: a list of instruction addresses which could be executed next. If the instruction stored at the PC is a branch instruction, the list contains the addresses of the next instruction and the branch destination; otherwise, it contains only the address of the next instruction. The follow-set is then used to replace each possible following instruction with a breakpoint instruction. The original instructions need not be saved; they remain in their unaltered state in the text file. The `startstop` builtin writes the 'startstop' message to the `proc` control file for the process named `pid`. The target process executes until some condition causes it to enter the kernel, in this case, the execution of a breakpoint. When the process blocks, the debugger regains control and invokes the ACID library function `stopped` which reports the address and cause of the blockage. The `startstop` function completes and returns to the `step` function where the follow-set is used to replace the breakpoints placed earlier. Finally, if the address of the original PC contained a breakpoint, it is replaced.

Notice that this approach to process control is inherently portable; the ACID code is shared by the debuggers for all architectures. ACID variables and builtin functions provide a transparent interface to architecture-dependent values and functions. Here the breakpoint value and format are referenced through ACID variables and the follow primitive masks the differences in the underlying instruction set.

The next function, similar to the *dbx* command of the same name, is a simpler example. This function steps through a single source statement but steps over function calls.

```
defn next()
{
    local sp, bound;

    sp = *SP;                // save starting SP
    bound = fnbound(*PC);    // begin & end of function
    stmt();                  // step 1 statement
    pc = *PC;
    if pc >= bound[0] && pc < bound[1] then
        return {};

    while (pc < bound[0] || pc > bound[1]) && sp >= *SP do {
        step();
        pc = *PC;
    }
    src(*PC);
}
```

The next function starts by saving the current stack pointer in a local variable. It then uses the ACID library function *fnbound* to return the addresses of the first and last instructions in the current function in a list. The *stmt* function executes a single source statement and then uses *src* to print a few lines of source around the new PC. If the new value of the PC remains in the current function, *next* returns. When the executed statement is a function call or a return from a function, the new value of the PC is outside the bounds calculated by *fnbound* and the test of the while loop is evaluated. If the statement was a return, the new value of the stack pointer is greater than the original value and the loop completes without execution. Otherwise, the loop is entered and instructions are continually executed until the value of the PC is between the bounds calculated earlier. At that point, execution ceases and a few lines of source in the vicinity of the PC are printed.

ACID provides concise and elegant expression for control and manipulation of target programs. These examples demonstrate how a few well-chosen primitives can be combined to create a rich debugging environment.

10. Dealing With Multiple Architectures

A single binary of ACID may be used to debug a program running on any of the five processor architectures supported by Plan 9. For example, Plan 9 allows a user on a MIPS to import the *proc* file system from an i486-based PC and remotely debug a program executing on that processor.

Two levels of abstraction provide this architecture independence. On the lowest level, a Plan 9 library supplies functions to decode the file header of the program being debugged and select a table of system parameters and a jump vector of architecture-dependent functions based on the magic number. Among these functions are byte-order-independent access to memory and text files, stack manipulation, disassembly, and floating point number interpretation. The second level of abstraction is supplied by ACID. It consists of primitives and approximately 200 lines of architecture-dependent ACID library code that interface the interpreter to the architecture-dependent library. This layer performs functions such as mapping register names to memory locations, supplying breakpoint values and sizes, and converting processor specific data to ACID data types. An example of the latter is the stack trace function *strace*, which uses the stack traversal functions in the architecture-dependent library to construct a list of lists describing the context of a process. The first level of list selects each function in the trace; subordinate lists contain the names and values of parameters and local variables of the functions. ACID commands and library functions that manipulate and display process state information operate on the list representation and are independent of the underlying architecture.

11. ALEF Runtime

ALEF is a concurrent programming language designed specifically for systems programming that supports both shared variable and message passing paradigms. ALEF borrows the C expression syntax but implements a substantially different type system. The language provides a rich set of exception handling, process management, and synchronization primitives which rely on a runtime system. ALEF program bugs are often deadlocks, synchronization failures, or non-termination caused by locks being held incorrectly. In such cases, a process stalls deep in the runtime code and it is clearly unreasonable to expect a programmer using the language to understand the detailed internal semantics of the runtime support functions.

Instead, there is an ALEF support library, coded in ACID, that allows the programmer to interpret the program state in terms of ALEF operations. Consider the example of a multi-process program stalling because of improper synchronization. A stack trace of the program indicates that it is waiting for an event in some obscure ALEF runtime synchronization function. The function itself is irrelevant to the programmer; of greater importance is the identity of the unfulfilled event. Commands in the ALEF support library decode the runtime data structures and program state to report the cause of the blockage in terms of the high-level operations available to the ALEF programmer. Here, the ACID language acts as a communications medium between ALEF implementer and ALEF user.

12. Parallel Debugging

The central issue in parallel debugging is how the debugger is multiplexed between the processes comprising the program. ACID has no intrinsic model of process partitioning; it only assumes that parallel programs share a symbol table, though they need not share memory. The `setproc` primitive attaches the debugger to a running process associated with the process ID supplied as its argument and assigns that value to the global variable `pid`, thereby allowing simple rotation among a group of processes. Further, the stack trace primitive is driven by parameters specifying a unique process context, so it is possible to examine the state of cooperating processes without switching the debugger focus from the process of interest. Since ACID is inherently extensible and capable of dynamic interaction with subordinate processes, the programmer can define ACID commands to detect and control complex interactions between processes. In short, the programmer is free to specify how the debugger reacts to events generated in specific threads of the program.

The support for parallel debugging in ACID depends on a crucial kernel modification: when the text segment of a program is written (usually to place a breakpoint), the segment is cloned to prevent other threads from encountering the breakpoint. Although this incurs a slight performance penalty, it is of little importance while debugging.

13. Communication Between Tools

Like the other Plan 9 compilers, the ALEF compiler does not embed detailed type information in the symbol table of an executable file. However, it does accept a command line option causing it to emit descriptions of complex data types (e.g., aggregates and abstract data types) to an auxiliary file. The vehicle for expressing this information is ACID source code. When an ACID debugging session is subsequently started, that file is loaded with the other ACID libraries.

For each complex object in the ALEF program the compiler generates three pieces of ACID code. The first is a table describing the size and offset of each member of the complex data type. Following is an ACID function, named the same as the object, that formats and prints each member. Finally, ACID declarations associate the ALEF program variables of a type with the functions to print them. The three forms of declaration are shown in the following example:

```
struct Bitmap {
    Rectangle    0 r;
    Rectangle   16 clipr;
    'D'         32 ldepth;
    'D'         36 id;
    'X'         40 cache;
};
```



```

defn
Bitmap(addr) {
    complex Bitmap addr;
    print("Rectangle r {\n");
    Rectangle(addr.r);
    print("}\n");
    print("Rectangle clipr {\n");
    Rectangle(addr.clipr);
    print("}\n");
    print(" ldepth  ", addr.ldepth, "\n");
    print(" id      ", addr.id, "\n");
    print(" cache   ", addr.cache, "\n");
};

complex Bitmap darkgrey;
complex Bitmap Window_settag:b;

```

The struct declaration specifies decoding instructions for the complex type named `Bitmap`. Although the syntax is superficially similar to a C structure declaration, the semantics differ markedly: the C declaration specifies a layout, while the ACID declaration tells how to decode it. The declaration specifies a type, an offset, and name for each member of the complex object. The type is either the name of another complex declaration, for example, `Rectangle`, or a format code. The offset is the number of bytes from the start of the object to the member and the name is the member's name in the ALEF declaration. This type description is a close match for C and ALEF, but is simple enough to be language independent.

The `Bitmap` function expects the address of a `Bitmap` as its only argument. It uses the decoding information contained in the `Bitmap` structure declaration to extract, format, and print the value of each member of the complex object pointed to by the argument. The ALEF compiler emits code to call other ACID functions where a member is another complex type; here, `Bitmap` calls `Rectangle` to print its contents.

The complex declarations associate ALEF variables with complex types. In the example, `darkgrey` is the name of a global variable of type `Bitmap` in the ALEF program being debugged. Whenever the name `darkgrey` is evaluated by ACID, it automatically calls the `Bitmap` function with the address of `darkgrey` as the argument. The second complex declaration associates a local variable or parameter named `b` in function `Window_settag` with the `Bitmap` complex data type.

ACID borrows the C operators `.` and `->` to access the decoding parameters of a member of a complex type. Although this representation is sufficiently general for describing the decoding of both C and ALEF complex data types, it may prove too restrictive for target languages with more complicated type systems. Further, the assumption that the compiler can select the proper ACID format code for each basic type in the language is somewhat naive. For example, when a member of a complex type is a pointer, it is assigned a hexadecimal type code; integer members are always assigned a decimal type code. This heuristic proves inaccurate when an integer field is a bit mask or set of bit flags which are more appropriately displayed in hexadecimal or octal.

14. Code Verification

ACID's ability to dynamically interact with an executing program allows passive test and verification of the target program. For example, a common concern is leak detection in programs using `malloc`. Of interest are two items: finding memory that was allocated but never freed and detecting bad pointers passed to `free`. An auxiliary ACID library contains ACID functions to monitor the execution of a program and detect these faults, either as they happen or in the automated post-mortem analysis of the memory arena. In the following example, the `sort` command is run under the control of the ACID memory leak library.

```

helix% acid -l malloc /bin/sort
/bin/sort: mips plan 9 executable
/lib/acid/port
/lib/acid/mips
/lib/acid/malloc
acid: go()
now
is
the
time
<ctrl-d>
is
now
the
time
27680 : breakpoint      _exits+0x4      MOVW      $0x8,R1
acid:

```

The go command creates a process and plants breakpoints at the entry to malloc and free. The program is then started and continues until it exits or stops. If the reason for stopping is anything other than the breakpoints in malloc and free, ACID prints the usual status information and returns to the interactive prompt.

When the process stops on entering malloc, the debugger must capture and save the address that malloc will return. After saving a stack trace so the calling routine can be identified, it places a breakpoint at the return address and restarts the program. When malloc returns, the breakpoint stops the program, allowing the debugger to grab the address of the new memory block from the return register. The address and stack trace are added to the list of outstanding memory blocks, the breakpoint is removed from the return point, and the process is restarted.

When the process stops at the beginning of free, the memory address supplied as the argument is compared to the list of outstanding memory blocks. If it is not found an error message and a stack trace of the call is reported; otherwise, the address is deleted from the list.

When the program exits, the list of outstanding memory blocks contains the addresses of all blocks that were allocated but never freed. The leak library function traverses the list producing a report describing the allocated blocks.

```

acid: leak()
Lost a total of 524288 bytes from:
    malloc() malloc.c:32 called from dofile+0xe8 sort.c:217
    dofile() sort.c:190 called from main+0xac sort.c:161
    main() sort.c:128 called from _main+0x20 main9.s:10
Lost a total of 64 bytes from:
    malloc() malloc.c:32 called from newline+0xfc sort.c:280
    newline() sort.c:248 called from dofile+0x110 sort.c:222
    dofile() sort.c:190 called from main+0xac sort.c:161
    main() sort.c:128 called from _main+0x20 main9.s:10
Lost a total of 64 bytes from:
    malloc() malloc.c:32 called from realloc+0x14 malloc.c:129
    realloc() malloc.c:123 called from buildkey+0x358 sort.c:1388
    buildkey() sort.c:1345 called from newline+0x150 sort.c:285
    newline() sort.c:248 called from dofile+0x110 sort.c:222
    dofile() sort.c:190 called from main+0xac sort.c:161
    main() sort.c:128 called from _main+0x20 main9.s:10
acid: refs()
data...bss...stack...
acid: leak()
acid:

```

The presence of a block in the allocation list does not imply it is there because of a leak; for instance, it may have been in use when the program terminated. The refs() library function scans the data, bss, and stack segments of the process looking for pointers into the allocated blocks. When one is found, the block is deleted from the outstanding block list. The leak function is used again to report the blocks remaining allocated and unreferenced. This strategy proves effective in detecting disconnected (but non-circular) data structures.

The leak detection process is entirely passive. The program is not specially compiled and the source code is not required. As with the ACID support functions for the ALEF runtime environment, the author of the library routines has encapsulated the functionality of the library interface in ACID code. Any programmer may then check a program's use of the library routines without knowledge of either implementation. The performance impact of running leak detection is great (about 10 times slower), however it has not prevented interactive programs like *sam* and the 8½ window system from being tested.

15. Code Coverage

Another common component of software test uses *coverage* analysis. The purpose of the test is to determine which paths through the code have not been executed while running the test suite. This is usually performed by a combination of compiler support and a reporting tool run on the output generated by statements compiled into the program. The compiler emits code that logs the progress of the program as it executes basic blocks and writes the results to a file. The file is then processed by the reporting tool to determine which basic blocks have not been executed.

ACID can perform the same function in a language independent manner without modifying the source, object or binary of the program. The following example shows *ls* being run under the control of the ACID coverage library.

```
philw-helix% acid -l coverage /bin/ls
/bin/ls: mips plan 9 executable
/lib/acid/port
/lib/acid/mips
/lib/acid/coverage
acid: coverage()
acid
newstime
profile
tel
wintool
2: (error) msg: pid=11419 startstop: process exited
acid: analyse(ls)
ls.c:102,105
    102:                return 1;
    103:            }
    104:            if(db[0].qid.path&CHDIR && dflag==0){
    105:                output();
ls.c:122,126
    122:                memmove(dirbuf+ndir, db, sizeof(Dir));
    123:                dirbuf[ndir].prefix = 0;
    124:                p = utfrrune(s, '/');
    125:                if(p){
    126:                    dirbuf[ndir].prefix = s;
```

The coverage function begins by looping through the text segment placing breakpoints at the entry to each basic block. The start of each basic block is found using the ACID builtin function *follow*. If the list generated by *follow* contains more than one element, then the addresses mark the start of basic blocks. A breakpoint is placed at each address to detect entry into the block. If the result of *follow* is a single address then no action is taken, and the next address is considered. ACID maintains a list of breakpoints already in place and avoids placing duplicates (an address may be the destination of several branches).

After placing the breakpoints the program is set running. Each time a breakpoint is encountered ACID deletes the address from the breakpoint list, removes the breakpoint from memory and then restarts the program. At any instant the breakpoint list contains the addresses of basic blocks which have not been executed. The *analyse* function reports the lines of source code bounded by basic blocks whose addresses have not been deleted from the breakpoint list. These are the basic blocks which have not been executed. Program performance is almost unaffected since each breakpoint is executed only once and then removed.

The library contains a total of 128 lines of ACID code. An obvious extension of this algorithm could be used to provide basic block profiling.

16. Conclusion

ACID has two areas of weakness. As with other language-based tools like *awk*, a programmer must learn yet another language to step beyond the normal debugging functions and use the full power of the debugger. Second, the command line interface supplied by the *yacc* parser is inordinately clumsy. Part of the problem relates directly to the use of *yacc* and could be circumvented with a custom parser. However, structural problems would remain: ACID often requires too much typing to execute a simple command. A debugger should prostitute itself to its users, doing whatever is wanted with a minimum of encouragement; commands should be concise and obvious. The language interface is more consistent than an ad hoc command interface but is clumsy to use. Most of these problems are addressed by an Acme interface which is under construction. This should provide the best of both worlds: graphical debugging and access to the underlying acid language when required.

The namespace clash between ACID variables, keywords, program variables, and functions is unavoidable. Although it rarely affects a debugging session, it is annoying when it happens and is sometimes difficult to circumvent. The current renaming scheme is too crude; the new names are too hard to remember.

ACID has proved to be a powerful tool whose applications have exceeded expectations. Of its strengths, portability, extensibility and parallel debugging support were by design and provide the expected utility. In retrospect, its use as a tool for code test and verification and as a medium for communicating type information and encapsulating interfaces has provided unanticipated benefits and altered our view of the debugging process.

Manuals for both ACID and ALEF are available in the Plan 9 manual or by anonymous FTP from `research.att.com` in the directory `dist/plan9man`.

17. Acknowledgments

Bob Flandrena was the first user and helped prepare the paper. Rob Pike endured three buggy ALEF compilers and a new debugger in a single sitting.

18. References

- [Pike90] R. Pike, D. Presotto, K. Thompson, H. Trickey, "Plan 9 from Bell Labs", *UKUUG Proc. of the Summer 1990 Conf. , London, England, 1990*
- [Gol93] M. Golan, D. Hanson, "DUEL -- A Very High-Level Debugging Language", *USENIX Proc. of the Winter 1993 Conf. , San Diego, CA, 1993*
- [Lin90] M. A. Linton, "The Evolution of DBX", *USENIX Proc. of the Summer 1990 Conf. , Anaheim, CA, 1990*
- [Stal91] R. M. Stallman, R. H. Pesch, "Using GDB: A guide to the GNU source level debugger", *Technical Report, Free Software Foundation, Cambridge, MA, 1991*
- [Win93] P. Winterbottom, "ALEF reference Manual", *The Plan 9 Manual, AT&T Bell Laboratories, Murray Hill, NJ, 1993*
- [Pike93] Rob Pike, "Acme: A User Interface for Programmers", *USENIX Proc. of the Winter 1994 Conf. , San Francisco, CA*
- [Ols90] Ronald A. Olsson, Richard H. Crawford, and W. Wilson Ho. , "Dalek: A GNU, improved programmable debugger", *USENIX Proc. of the Summer 1990 Conf. , Anaheim, CA*
- [May92] Paul Maybee, "NeD: The Network Extensible Debugger" *USENIX Proc. of the Summer 1992 Conf. , San Antonio, TX*
- [Aral] Ziya Aral, Ilya Gertner, and Greg Schaffer. , "Efficient debugging primitives for multiprocessors", *In Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems. , SIGPLAN notices Nr. 22, May 1989.*

Phil Winterbottom is a member of technical staff at AT&T Bell Laboratories. He works in the areas of languages, operating systems and networks for distributed computing. He is one of the authors of Plan 9.

Acme: A User Interface for Programmers

Rob Pike

*AT&T Bell Laboratories
Murray Hill, New Jersey 07974*

ABSTRACT

A hybrid of window system, shell, and editor, Acme gives text-oriented applications a clean, expressive, and consistent style of interaction. Traditional window systems support interactive client programs and offer libraries of pre-defined operations such as pop-up menus and buttons to promote a consistent user interface among the clients. Acme instead provides its clients with a fixed user interface and simple conventions to encourage its uniform use. Clients access the facilities of Acme through a file system interface; Acme is in part a file server that exports device-like files that may be manipulated to access and control the contents of its windows. Written in a concurrent programming language, Acme is structured as a set of communicating processes that neatly subdivide the various aspects of its tasks: display management, input, file server, and so on.

Acme attaches distinct functions to the three mouse buttons: the left selects text; the middle executes textual commands; and the right combines context search and file opening functions to integrate the various applications and files in the system.

Acme works well enough to have developed a community that uses it exclusively. Although Acme discourages the traditional style of interaction based on typescript windows—teletypes—its users find Acme's other services render typescripts obsolete.

History and motivation

The usual typescript style of interaction with Unix and its relatives is an old one. The typescript—an intermingling of textual commands and their output—originates with the scrolls of paper on teletypes. The advent of windowed terminals has given each user what amounts to an array of teletypes, a limited and unimaginative use of the powers of bitmap displays and mice. Systems like the Macintosh that do involve the mouse as an integral part of the interaction are geared towards general users, not experts, and certainly not programmers. Software developers, at least on time-sharing systems, have been left behind.

Some programs have mouse-based editing of text files and typescripts; ones I have built include the window systems *mux* [Pike88] and *8½* [Pike91] and the text editor *Sam* [Pike87]. These have put the programmer's mouse to some productive work, but not wholeheartedly. Even experienced users of these programs often retype text that could be grabbed with the mouse, partly because the menu-driven interface is imperfect and partly because the various pieces are not well enough integrated.

Other programs—EMACS [Stal93] is the prime example—offer a high degree of integration but with a user interface built around the ideas of cursor-addressed terminals that date from the 1970's. They are still keyboard-intensive and dauntingly complex.

The most ambitious attempt to face these issues was the Cedar system, developed at Xerox [Swei86]. It combined a new programming language, compilers, window system, even microcode—a complete system—to construct a productive, highly integrated and interactive environment for experienced users of compiled languages. Although successful internally, the system was so large and so tied to specific hardware that it never fledged.

Mail Newcol Kill Exit	
New Cut Paste Snarf Delcol	
/usr/rob/src/acme/dat.h Del Snarf Xfid	
<pre> adt Xfid{ extern Fcall; extern Xfid *next; extern chan(void*)(Xfid*) c; void ctl(Xfid); </pre>	
/usr/rob/src/acme/acme.l Del Snarf Undo Look mk cxfidalloc	
<pre> xfidalloc(task(chan(int) *exit0, chan(int) *exit1) { Xfid *xfree, *x; xfree = nil; for(;;){ alt{ case <-*exit0: *exit1 <= 1; return; case <-cxfidalloc x = xfree; if(x) xfree = x->next; else{ x = emalloc(sizeof(Xfid)); alloc x->c; task x->ctl0; } </pre>	
Acid/396/stk Del Snarf Send Delete	
<pre> Rendez_Sleep(*Rendez_Sleep:r=0x00045170,*Rendez_Sleep:bool=0x000 45178,*Rendez_Sleep:t=0x00000000) called from ALEF_rcvint+0x184 /sys/src/alef/lib/mips/alef.l:582 ALEF_rcvint(*ALEF_rcvint:c=0x0004512c) called from timeproc+0x1cc /usr/rob/src/acme/time.l:63 timeproc0 called from ALEF_linksp+0x18 /sys/src/alef/lib/mips/alefasm.s:13 </pre>	
/usr/rob/src/acme/time.l Del Snarf Look	
<pre> { if(nt==0 ?ctimer){ x = <-ctimer; if(nt == na){ na += 10; t = erealloc(t, na*sizeof(Timer*)); } t[nt++] = x; } </pre>	
New Cut Paste Snarf Delcol	
/acme/acid/guide Del Snarf Undo Put Get	
Acid pid	
Acid -l alef -l /usr/rob/src/acme/acid 396	
/acme/bin/guide Del Snarf	
/acme/edit/guide Del Snarf	
/acme/mail/guide Del Snarf	
Mail /mail/box/\$user/stored	
mkbox /mail/box/\$user/new_box	
mail -# someaddress	
Mail/mbox/ Del Snarf Look Put Mail	
28 bobf Wed Oct 13 09:47	
26 bwk Thu Oct 12 15:32	
25 adb Wed Oct 12 12:21	
24 dmr Thu May 20 07:26	
/usr/rob/src/acme/ Del Snarf Get	
acid dat.h fsys.l rows.l wind.l	
acme.l disk.l look.l scrl.l xfid.l	
addr.l exec.l man text.l	
buff.l file.l mkfile time.l	
cols.l fns.h regx.l util.l	
adict/oed/ Del Snarf	
futtock	
adict/oed/futtock Del Snarf Prev Next	
futtock ('fatək). Naut. Also 8 foot-hook. [prob., as al	
ready	
suggested in quot. 1644, a pronunciation of foot-ho	
ok (see quot. 1769).]	
1. One of the middle timbers of the frame of a ship.	
/acme/acid/-Acid Del Snarf Send Delete	
acid: pchan(*ctimer)	
No sender	
Receiver waiting: buffer 0x000471dc	
acid:	
/usr/rob/src/acme/+Errors Del Snarf	
acme.l:439: syntax error, near symbol 'case'	
acme.l:445: syntax error, near symbol 'break'	
mk: val -w acme.l : exit status=rc 509:val 511:errors	
mk 508:error	

Figure 1. A small Acme screen—normally it runs on a larger display—demonstrating some of the details discussed in the text. The right column contains some guide files, a mailbox presented by Acme's mail program, the columnated display of files in Acme's own source directory, a couple of windows from the OED browser, a debugger window, and an error window showing diagnostics from a compilation. The left column holds a couple of source files (dat.h and acme.l), another debugger window displaying a stack trace, and a third source file (time.l). Time.l was opened from the debugger by clicking the right mouse button on a line in the stack window; the mouse cursor landed on the offending line of acme.l after a click on the compiler diagnostic.

Cedar was, however, the major inspiration for Oberon [Wirt89], a system of similar scope but much smaller scale. Through careful selection of Cedar's ideas, Oberon shows that its lessons can be applied to a small, coherent system that can run efficiently on modest hardware. In fact, Oberon probably errs too far towards simplicity: a single-process system with weak networking, it seems an architectural throwback.

Acme is a new program, a combined window system, editor, and shell, that applies some of the ideas distilled by Oberon. Where Oberon uses objects and modules within a programming language (also called Oberon), Acme uses files and commands within an existing operating system (Plan 9). Unlike Oberon, Acme has does not yet have support for graphical output, just text. At least for now, the work on Acme has concentrated on producing the smoothest user interface possible for a programmer at work.

The rest of this paper describes Acme's interface, explains how programs can access it, compares it to existing systems, and finally presents some unusual aspects of its implementation.

```

/usr/rob/src/acme/acme.l Del Snarf Undo Put Get Look | mk cxfidalloc
xfidalloctask(chan(int) *exit0, chan(int) *exit1)
{
    Xfid *xfree, *x;
    xfree = nil;
    for(;;) alt{
        case <-*exit0:
            *exit1 <-= 1;
            return;
        case <-cxfidalloc:
            x = xfree;
            if(x)
                xfree = x->next;
            else{
                x = emalloc(sizeof(Xfid));
                alloc x->c;
                task x->ctl0;
            }
            cxfidalloc <-= x;
            break;
        case x = <-cxfidfree:
            x->next = xfree;
            xfree = x;
            break;
    }
}

```

Figure 2. An Acme window showing a section of code. The upper line of text is the tag containing the file name, relevant commands, and a scratch area (right of the vertical bar); the lower portion of the window is the body, or contents, of the file. Here the scratch area contains a command for the middle button (mk) and a word to search for with the right button (cxfidalloc). The user has just clicked the right button on cxfidalloc and Acme has searched for the word, highlighted it, and moved the mouse cursor there. The file has been modified: the center of the layout box is black and the command Put appears in the tag.

User interface

Acme windows are arrayed in columns (Figure 1) and are used more dynamically than in an environment like X Windows or 8½ [Sche86, Pike91]. The system frequently creates them automatically and the user can order a new one with a single mouse button click. The initial placement of a new window is determined automatically, but the user may move an existing window anywhere by clicking or dragging a *layout box* in the upper left corner of the window.

Acme windows have two parts: a *tag* holding a single line of text, above a *body* holding zero or more lines (Figure 2). The body typically contains an image of a file being edited or the editable output of a program, analogous to an EMACS shell window. The tag contains the name of the window (usually the name of the associated file or directory), some built-in commands, and a scratch area to hold arbitrary text. If a window represents a directory, the name in the tag ends with a slash and the body contains a list of the names of the files in the directory. Finally, each non-empty body holds a scroll bar at the left of the text.

Each column of windows also has a layout box and a tag. The tag has no special meaning, although Acme pre-loads it with a few built-in commands. There is also a tag across the whole display, also loaded with helpful commands and a list of active processes started by Acme.

Typing with the keyboard and selecting with the left button are as in many other systems, including the Macintosh, 8½, and Sam. The middle and right buttons are used, somewhat like the left button, to ‘sweep’ text, but the indicated text is treated in a way that depends on the text’s location—*context*—as well as its content. This context, based on the directory of the file containing the text, is a central component of Acme’s style of interaction.

Acme has no single notion of ‘current directory’. Instead, every command, file name, action, and so on is interpreted or executed in the directory named by the tag of the window containing the command. For example, the string `mammals` in a window labeled `/lib/` or `/lib/insects` will be interpreted as the file name `/lib/mammals` if such a file exists.

Throughout Acme, the middle mouse button is used to execute commands and the right mouse button is used to locate and select files and text. Even when there are no true files on which to operate—for example when editing

mail messages—Acme and its applications use consistent extensions of these basic functions. This idea is as vital to Acme as icons are to the Macintosh.

The middle button executes commands: text swept with the button pressed is underlined; when the button is released, the underline is removed and the indicated text is executed. A modest number of commands are recognized as built-ins: words like Cut, Paste, and New name functions performed directly by Acme. These words often appear in tags to make them always available, but the tags are not menus: any text anywhere in Acme may be a command. For example, in the tag or body of any window one may type Cut, select it with the left button, use the middle button to execute it, and watch it disappear again.

If the middle button indicates a command that is not recognized as a built-in, it is executed in the directory named by the tag of the window holding the text. Also, the file to be executed is searched for first in that directory. Standard input is connected to `/dev/null`, but standard and error outputs are connected to an Acme window, created if needed, called `dir/+Errors` where `dir` is the directory of the window. (Programs that need interactive input use a different interface, described below.) A typical use of this is to type `mk` (Plan 9's make) in the scratch area in the tag of a C source window, say `/sys/src/cmd/sam/regexp.c`, and execute it. Output, including compiler errors, appears in the window labeled `/sys/src/cmd/sam/+Errors`, so file names in the output are associated with the windows and directory holding the source. The `mk` command remains in the tag, serving as a sort of menu item for the associated window.

Like the middle button, the right button is used to indicate text by sweeping it out. The indicated text is not a command, however, but the argument of a generalized search operator. If the text, perhaps after appending it to the directory of the window containing it, is the name of an existing file name, Acme creates a new window to hold the file and reads it in. It then moves the mouse cursor to that window. If the file is already loaded into Acme, the mouse motion happens but no new window is made. For example, indicating the string `sam.h` in

```
#include "sam.h"
```

in a window on the file `/sys/src/cmd/sam/regexp.c` will open the file `/sys/src/cmd/sam/sam.h`.

If the file name is followed immediately by a colon and a legal address in Sam notation (for example a line number or a regular expression delimited in slashes or a comma-separated compound of such addresses), Acme highlights the target of that address in the file and places the mouse there. One may jump to line 27 of `dat.h` by indicating with the right button the text `dat.h:27`. If the file is not already open, Acme loads it. If the file name is null, for example if the indicated string is `:/^main/`, the file is assumed to be that of the window containing the string. Such strings, when typed and evaluated in the tag of a window, amount to context searches.

If the indicated text is not the name of an existing file, it is taken to be literal text and is searched for in the body of the window containing the text, highlighting the result as if it were the result of a context search.

For the rare occasion when a file name *is* just text to search for, it can be selected with the left button and used as the argument to a built-in `Look` command that always searches for literal text.

Nuances and heuristics

A user interface should not only provide the necessary functions, it should also *feel* right. In fact, it should almost not be felt at all; when one notices a user interface, one is distracted from the job at hand [Pike88]. To approach this invisibility, some of Acme's properties and features are there just to make the others easy to use. Many are based on a fundamental principle of good design: let the machine do the work.

Acme tries to avoid needless clicking and typing. There is no 'click-to-type', eliminating a button click. There are no pop-up or pull-down menus, eliminating the mouse action needed to make a menu appear. The overall design is intended to make text on the screen useful without copying or retyping; the ways in which this happens involve the combination of many aspects of the interface.

Acme tiles its windows and places them automatically to avoid asking the user to place and arrange them. For this policy to succeed, the automatic placement must behave well enough that the user is usually content with the location of a new window. The system will never get it right all the time, but in practice most windows are used at least for a while where Acme first places them. There have been several complete rewrites of the heuristics for placing a new window, and with each rewrite the system became noticeably more comfortable. The rules are as follows, although they are still subject to improvement. The window appears in the 'active' column, that most recently used

for typing or selecting. Executing and searching do not affect the choice of active column, so windows of commands and such do not draw new windows towards them, but rather let them form near the targets of their actions. Output (error) windows always appear towards the right, away from edited text, which is typically kept towards the left. Within the column, several competing desires are balanced to decide where and how large the window should be: large blank spaces should be consumed; existing text should remain visible; existing large windows should be divided before small ones; and the window should appear near the one containing the action that caused its creation.

Acme binds some actions to chords of mouse buttons. These include Cut and Paste so these common operations can be done without moving the mouse. Another is a way to apply a command in one window to text (often a file name) in another, avoiding the actions needed to assemble the command textually.

Another way Acme avoids moving the mouse is instead to move the cursor to where it is likely to be used next. When a new window is made, Acme moves the cursor to the new window; in fact, to the selected text in that window. When the user deletes a newly made window, the cursor is returned to the point it was before the window was made, reducing the irritation of windows that pop up to report annoying errors.

When a window is moved, Acme moves the cursor to the layout box in its new place, to permit further adjustment without moving the mouse. For example, when a click of the left mouse button on the layout box grows the window, the cursor moves to the new location of the box so repeated clicks, without moving the mouse, continue to grow it.

Another form of assistance the system can offer is to supply precision in pointing the mouse. The best-known form of this is 'double-clicking' to select a word rather than carefully sweeping out the entire word. Acme provides this feature, using context to decide whether to select a word, line, quoted string, parenthesized expression, and so on. But Acme takes the idea much further by applying it to execution and searching. A *single* click, that is, a null selection, with either the middle or right buttons, is expanded automatically to indicate the appropriate text containing the click. What is appropriate depends on the context.

For example, to execute a single-word command such as Cut, it is not necessary to sweep the entire word; just clicking the button once with the mouse pointing at the word is sufficient. 'Word' means the largest string of likely file name characters surrounding the location of the click: click on a file name, run that program. On the right button, the rules are more complicated because the target of the click might be a file name, file name with address, or just plain text. Acme examines the text near the click to find a likely file name; if it finds one, it checks that it names an existing file (in the directory named in the tag, if the name is relative) and if so, takes that as the result, after extending it with any address that may be present. If there is no file with that name, Acme just takes the largest alphanumeric string under the click. The effect is a natural overloading of the button to refer to plain text as well as file names.

First, though, if the click occurs over the left-button-selected text in the window, that text is taken to be what is indicated. This makes it easy to skip through the occurrences of a string in a file: just click the right button on some occurrence of the text in the window (perhaps after typing it in the tag) and click once for each subsequent occurrence. It isn't even necessary to move the mouse between clicks; Acme does that. To turn a complicated command into a sort of menu item, select it: thereafter, clicking the middle button on it will execute the full command.

As an extra feature, Acme recognizes file names in angle brackets <> as names of files in standard directories or include files, making it possible for instance to look at <stdio.h> with a single click.

Here's an example to demonstrate how the actions and defaults work together. Assume /sys/src/cmd/sam/regexp.c is open and has been edited. We write it (execute Put in the tag; once the file is written, Acme removes the word from the tag) and type mk in the tag. We execute mk and get some errors, which appear in a new window labeled /sys/src/cmd/sam/+Errors. The cursor moves automatically to that window. Say the error is

```
main.c:112: incompatible types on assignment to 'pattern'
```

We move the mouse slightly and click the right button at the left of the error message; Acme makes a new window, reads /sys/src/cmd/main.c into it, selects line 112 in and places the mouse there, right on the offending line.

Coupling to existing programs

Acme's syntax for file names and addresses makes it easy for other programs to connect automatically to Acme's capabilities. For example, the output of

```
grep -n variable *.ch]
```

can be used to help Acme step through the occurrences of a variable in a program; every line of output is potentially a command to open a file. The file names need not be absolute, either: the output appears in a window labeled with the directory in which `grep` was run, from which Acme can derive the full path names.

When necessary, we have changed the output of some programs, such as compiler error messages, to match Acme's syntax. Some might argue that it shouldn't be necessary to change old programs, but sometimes programs need to be updated when systems change, and consistent output benefits people as well as programs. A historical example is the retrofitting of standard error output to the early Unix programs when pipes were invented.

Another change was to record full path names in the symbol table of executables, so line numbers reported by the debugger are absolute names that may be used directly by Acme; it's not necessary to run the debugger in the source directory. (This aids debugging even without Acme.)

A related change was to add lines of the form

```
#pragma src "/sys/src/libregexp"
```

to header files; coupled with Acme's ability to locate a header file, this provides a fast, keyboardless way to get the source associated with a library.

Finally, Acme directs the standard output of programs it runs to windows labeled by the directory in which the program is run. Acme's splitting of the output into directory-labeled windows is a small feature that has a major effect: local file names printed by programs can be interpreted directly by Acme. By indirectly coupling the output of programs to the input, it also simplifies the management of software that occupies multiple directories.

Coupling to new programs

Like many Plan 9 programs, Acme offers a programmable interface to other programs by acting as a file server. The best example of such a file server is the window system 8½ [Pike91], which exports files with names such as `screen`, `cons`, and `mouse` through which applications may access the I/O capabilities of the windows. 8½ provides a *distinct* set of files for each window and builds a private file name space for the clients running 'in' each window; clients in separate windows see distinct files with the same names (for example `/dev/mouse`). Acme, like the process file system [PPTTW93], instead associates each window with a directory of files; the files of each window are visible to any application. This difference reflects a difference in how the systems are used: 8½ tells a client what keyboard and mouse activity has happened in its window; Acme tells a client what changes that activity wrought on any window it asks about. Putting it another way, 8½ enables the construction of interactive applications; Acme provides the interaction for applications.

The root of Acme's file system is mounted using Plan 9 operations on the directory `/mnt/acme`. In that root directory appears a directory for each window, numbered with the window's identifier, analogous to a process identifier, for example `/mnt/acme/27`. The window's directory contains 6 files: `/mnt/acme/27/addr`, `body`, `ctl`, `data`, `event`, and `tag`. The `body` and `tag` files contain the text of the respective parts of the window; they may be read to recover the contents. Data written to these files is appended to the text; seeks are ignored. The `addr` and `data` files provide random access to the contents of the `body`. The `addr` file is written to set a character position within the `body`; the `data` file may then be read to recover the contents at that position, or written to change them. (The `tag` is assumed small and special-purpose enough not to need special treatment. Also, `addr` indexes by character position, which is not the same as byte offset in Plan 9's multi-byte character set [Pike93]). The format accepted by the `addr` file is exactly the syntax of addresses within the user interface, permitting regular expressions, line numbers, and compound addresses to be specified. For example, to replace the contents of lines 3 through 7, write the text

```
3,7
```

to the `addr` file, then write the replacement text to the `data` file. A zero-length write deletes the addressed text; further writes extend the replacement.

The control file, `ctl`, may be written with commands to effect actions on the window; for example the command

```
name /adm/users
```

sets the name in the tag of the window to `/adm/users`. Other commands allow deleting the window, writing it to a file, and so on. Reading the `ctl` file recovers a fixed-format string containing 5 textual numbers—the window identifier, the number of characters in the tag, the number in the body, and some status information—followed by the text of the tag, up to a newline.

The last file, `event`, is the most unusual. A program reading a window's event file is notified of all changes to the text of the window, and is asked to interpret all middle- and right-button actions. The data passed to the program is fixed-format and reports the source of the action (keyboard, mouse, external program, etc.), its location (what was pointed at or modified), and its nature (change, search, execution, etc.). This message, for example,

```
MI15 19 0 4 time
```

reports that actions of the mouse (M) inserted in the body (capital I) the 4 characters of `time` at character positions 15 through 19; the zero is a flag word. Programs may apply their own interpretations of searching and execution, or may simply reflect the events back to Acme, by writing them back to the `event` file, to have the default interpretation applied. Some examples of these ideas in action are presented below.

Notice that changes to the window are reported after the fact; the program is told about them but is not required to act on them. Compare this to a more traditional interface in which a program is told, for example, that a character has been typed on the keyboard and must then display and interpret it. Acme's style stems from the basic model of the system, in which any number of agents—the keyboard, mouse, external programs writing to data or body, and so on—may change the contents of a window. The style is efficient: many programs are content to have Acme do most of the work and act only when the editing is completed. An example is the Acme mail program, which can ignore the changes made to a message being composed and just read its body when asked to send it. A disadvantage is that some traditional ways of working are impossible. For example, there is no way 'to turn off echo': characters appear on the screen and are read from there; no agent or buffer stands between the keyboard and the display.

There are a couple of other files made available by Acme in its root directory rather than in the directory of each window. The text file `/mnt/acme/index` holds a list of all window names and numerical identifiers, somewhat analogous to the output of the `ps` command for processes. The most important, though, is `/mnt/acme/new`, a directory that makes new windows, similar to the `clone` directory in the Plan 9 network devices [Pres93]. The act of opening any file in `new` creates a new Acme window; thus the shell command

```
grep -n var *.c > /mnt/acme/new/body
```

places its output in the body of a fresh window. More sophisticated applications may open `new/ctl`, read it to discover the new window's identifier, and then open the window's other files in the numbered directory.

Acme-specific programs

Although Acme is in part an attempt to move beyond typescripts, they will probably always have utility. The first program written for Acme was therefore one to run a shell or other traditional interactive application in a window, the Acme analog of `xterm`. This program, `win`, has a simple structure: it acts as a two-way intermediary between Acme and the shell, cross-connecting the standard input and output of the shell to the text of the window. The style of interaction is modeled after `mux` [Pike88]: standard output is added to the window at the *output point*; text typed after the output point is made available on standard input when a newline is typed. After either of these actions, the output point is advanced. This is different from the working of a regular terminal, permitting cut-and-paste editing of an input line until the newline is typed. Arbitrary editing may be done to any text in the window. The implementation of `win`, using the `event`, `addr`, and `data` files, is straightforward. `Win` needs no code for handling the keyboard and mouse; it just monitors the contents of the window. Nonetheless, it allows Acme's full editing to be applied to shell commands. The division of labor between `win` and Acme contrasted with `xterm` and the X server demonstrates how much work Acme handles automatically. `Win` is implemented by a single source file 560 lines long and has no graphics code.

Win uses the middle and right buttons to connect itself in a consistent way with the rest of Acme. The middle button still executes commands, but in a style more suited to typescripts. Text selected with the middle button is treated as if it had been typed after the output point, much as a similar feature in `xterm` or `8½`, and therefore causes it to be 'executed' by the application running in the window. Right button actions are reflected back to Acme but refer to the appropriate files because win places the name of the current directory in the tag of the window. If the shell is running, a simple shell function replacing the `cd` command can maintain the tag as the shell navigates the file system. This means, for example, that a right button click on a file mentioned in an `ls` listing opens the file within Acme.

Another Acme-specific program is a mail-reader that begins by presenting, in a window, a listing of the messages in the user's mailbox, one per line. Here the middle and right button actions are modified to refer to mail commands and messages, but the change feels natural. Clicking the right button on a line creates a new window and displays the message there, or, if it's already displayed, moves the mouse to that window. The metaphor is that the mailbox is a directory whose constituent files are messages. The mail program also places some relevant commands in the tag lines of the windows; for example, executing the word `Reply` in a message's tag creates a new window in which to compose a message to the sender of the original; `Post` then dispatches it. In such windows, the addressee is just a list of names on the first line of the body, which may be edited to add or change recipients. The program also monitors the mailbox, updating the 'directory' as new messages arrive.

The mail program is as simple as it sounds; all the work of interaction, editing, and management of the display is done by Acme. The only difficult sections of the 1200 lines of code concern honoring the external protocols for managing the mailbox and connecting to `sendmail`.

One of the things Acme does not provide directly is a facility like Sam's command language to enable actions such as global substitution; within Acme, all editing is done manually. It is easy, though, to write external programs for such tasks. In this, Acme comes closer to the original intent of Oberon: a directory, `/acme/edit`, contains a set of tools for repetitive editing and a template or 'guide' file that gives examples of its use. Acme's editing guide, `/acme/edit/guide`, looks like this:

```
e file | x '/regexp/' | c 'replacement'
e file:'0,$' | x '/*word.*\n/' | p -n
e file | pipe command args ...
```

The syntax is reminiscent of Sam's command language, but here the individual one-letter commands are all stand-alone programs connected by pipes. Passed along the pipes are addresses, analogous to structural expressions in Sam terminology. The `e` command, unlike that of Sam, starts the process by generating the address (default dot, the highlighted selection) in the named files. The other commands are as in Sam: `p` prints the addressed text on standard output (the `-n` option is analogous to that of `grep`, useful in combination with the right mouse button), `x` matches a regular expression to the addressed (incoming) text, subdividing the text, `c` replaces the text, and so on. Thus, global substitution throughout a file, which would be expressed in Sam as

```
0,$ x/regexp/ c/replacement/
```

in Acme's editor becomes

```
e 'file:0,$' | x '/regexp/' | c 'replacement'
```

To use the Acme editing commands, open `/acme/edit/guide`, use the mouse and keyboard to edit one of the commands to the right form, and execute it with the middle button. Acme's context rules find the appropriate binaries in `/acme/edit` rather than `/bin`; the effect is to turn `/acme/edit` into a toolbox containing tools and instructions (the guide file) for their use. In fact, the source for these tools is also there, in the directory `/acme/edit/src`. This setup allows some control of the file name space for binary programs; not only does it group related programs, it permits the use of common names for uncommon jobs. For example, the single-letter names would be unwise in a directory in everyone's search path; here they are only visible when running editing commands.

In Oberon, such a collection would be called a *tool* and would consist of a set of entry points in a module and a menu-like piece of text containing representative commands that may be edited to suit and executed. There is, in fact, a tool called `Edit` in Oberon. To provide related functionality, Acme exploits the directory and file structure of the underlying system, rather than the module structure of the language; this fits well with Plan 9's file-oriented

philosophy. Such tools are central to the working of Oberon but they are less used in Acme, at least so far. The main reason is probably that Acme's program interface permits an external program to remain executing in the background, providing its own commands as needed (for example, the `Reply` command in the mail program); Oberon uses tools to implement such services because it must invoke a fresh program for each command. Also, Acme's better integration allows more basic functions to be handled internally; the right mouse button covers a lot of the basic utility of the editing tools in Oberon. Nonetheless, as more applications are written for Acme, many are sure to take this Oberon tool-like form.

Comparison with other systems

Acme's immediate ancestor is Help [Pike92], an experimental system written a few years ago as a first try at exploring some of Oberon's ideas in an existing operating system. Besides much better engineering, Acme's advances over Help include the actions of the right button (Help had nothing comparable), the ability to connect long-running programs to the user interface (Help had no analog of the `event` file), and the small but important change to split command output into windows labeled with the directory in which the commands run.

Most of Acme's style, however, derives from the user interface and window system of Oberon [Wirt89, Reis91]. Oberon includes a programming language and operating system, which Acme instead borrows from an existing system, Plan 9. When I first saw Oberon, in 1988, I was struck by the simplicity of its user interface, particularly its lack of menus and its elegant use of multiple mouse buttons. The system seemed restrictive, though—single process, single language, no networking, event-driven programming—and failed to follow through on some of its own ideas. For example, the middle mouse button had to be pointed accurately and the right button was essentially unused. Acme does follow through: to the basic idea planted by Oberon, it adds the ability to run on different operating systems and hardware, connection to existing applications including interactive ones such as shells and debuggers, support for multiple processes, the right mouse button's features, the default actions and context-dependent properties of execution and searching, and a host of little touches such as moving the mouse cursor that make the system more pleasant. At the moment, though, Oberon does have one distinct advantage: it incorporates graphical programs well into its model, an issue Acme has not yet faced.

Acme shares with the Macintosh a desire to use the mouse well and it is worth comparing the results. The mouse on the Macintosh has a single button, so menus are essential and the mouse must frequently move a long way to reach the appropriate function. An indication that this style has trouble is that applications provide keyboard sequences to invoke menu selections and users often prefer them. A deeper comparison is that the Macintosh uses pictures where Acme uses text. In contrast to pictures, text can be edited quickly, created on demand, and fine-tuned to the job at hand; consider adding an option to a command. It is also self-referential; Acme doesn't need menus because any text can be in effect a menu item. The result is that, although a Macintosh screen is certainly prettier and probably more attractive, especially to beginners, an Acme screen is more dynamic and expressive, at least for programmers and experienced users.

For its role in the overall system, Acme most resembles EMACS [Stal93]. It is tricky to compare Acme to EMACS, though, because there are many versions of EMACS and, since it is fully programmable, EMACS can in principle do anything Acme does. Also, Acme is much younger and therefore has not had the time to acquire as many features. The issue therefore is less what the systems can be programmed to do than how they are used. The EMACS versions that come closest to Acme's style are those that have been extended to provide a programming environment, usually for a language such as LISP [Alle92, Lucid92]. For richness of the existing interface, these EMACS versions are certainly superior to Acme. On the other hand, Acme's interface works equally well already for a variety of languages; for example, one of its most enthusiastic users works almost exclusively in Standard ML, a language nothing like C.

Where Acme excels is in the smoothness of its interface. Until recently, EMACS did not support the mouse especially well, and even with the latest version providing features such as 'extents' that can be programmed to behave much like Acme commands, many users don't bother to upgrade. Moreover, in the versions that provide extents, most EMACS packages don't take advantage of them.

The most important distinction is just that EMACS is fundamentally keyboard-based, while Acme is mouse-based.

People who try Acme find it hard to go back to their previous environment. Acme automates so much that to return to a traditional interface is to draw attention to the extra work it requires.

Concurrency in the implementation

Acme is about 8,000 lines of code in Alef, a concurrent object-oriented language syntactically similar to C [ALEF]. Acme's structure is a set of communicating processes in a single address space. One subset of the processes drives the display and user interface, maintaining the windows; other processes forward mouse and keyboard activity and implement the file server interface for external programs. The language and design worked out well; as explained elsewhere [Pike89, Reppy91, Gans93], user interfaces built with concurrent systems can avoid the clumsy top-level event loop typical of traditional interactive systems.

An example of the benefits of the multi-process style is the management of the state of open files held by clients of the file system interface. The problem is that some I/O requests, such as reading the event file, may block if no data is available, and the server must maintain the state of (possibly many) requests until data appears. For example, in 8½, a single-process window system written in C, pending requests were queued in a data structure associated with each window. After activity in the window that might complete pending I/O, the data structure was scanned for requests that could now finish. This structure did not fit well with the rest of the program and, worse, required meticulous effort to guarantee correct behavior under all conditions (consider raw mode, reads of partial lines, deleting a window, multibyte characters, etc.).

Acme instead creates a new dedicated process for each I/O request. This process coordinates with the rest of the system using Alef's synchronous communication; its state implicitly encodes the state of the I/O request and obviates the need for queuing. The passage of the request through Acme proceeds as follows.

Acme contains a file server process, F, that executes a read system call to receive a Plan 9 file protocol (9P) message from the client [AT&T92]. The client blocks until Acme answers the request. F communicates with an allocation process, M, to acquire an object of type Xfid ('executing fid'; fid is a 9P term) to hold the request. M sits in a loop (reproduced in Figure 2) waiting for either a request for a new Xfid or notification that an existing one has finished its task. When an Xfid is created, an associated process, X, is also made. M queues idle Xfids, allocating new ones only when the list is empty. Thus, there is always a pool of Xfids, some executing, some idle.

The Xfid object contains a channel, Xfid.c, for communication with its process; the unpacked message; and some associated functions, mostly corresponding to 9P messages such as Xfid.write to handle a 9P write request.

The file server process F parses the message to see its nature—open, close, read, write, etc. Many messages, such as directory lookups, can be handled immediately; these are responded to directly and efficiently by F without invoking the Xfid, which is therefore maintained until the next message. When a message, such as a write to the display, requires the attention of the main display process and interlocked access to its data structures, F enables X by sending a function pointer on Xfid.c. For example, if the message is a write, F executes

```
x->c <== Xfid.write;
```

which sends the address of Xfid.write on Xfid.c, waking up X.

The Xfid process, X, executes a simple loop:

```
void
Xfidctl(Xfid *x)
{
    for(;;){
        (*x->c)(x); /* receive and execute message */
        bflush(); /* synchronize bitmap display */
        cxfidfree <== x; /* return to free list */
    }
}
```

Thus X will wake up with the address of a function to call (here Xfid.write) and execute it; once that completes, it returns itself to the pool of free processes by sending its address back to the allocator.

Although this sequence may seem complicated, it is just a few lines of code and is in fact far simpler than the management of the I/O queues in 8½. The hard work of synchronization is done by the Alef run time system. Moreover, the code worked the first time, which cannot be said for the code in 8½.

Undo

Acme provides a general undo facility like that of Sam, permitting textual changes to be unwound arbitrarily. The implementation is superior to Sam's, though, with much higher performance and the ability to 'redo' changes.

Sam uses a multi-pass algorithm that builds a transcript of changes to be made simultaneously and then executes them atomically. This was thought necessary because the elements of a repetitive command such as a global substitution should all be applied to the same initial file and implemented simultaneously; forming the complete transcript before executing any of the changes avoids the cumbersome management of addresses in a changing file. Acme, however, doesn't have this problem; global substitution is controlled externally and may be made incrementally by exploiting an observation: if the changes are sorted in address order and executed in reverse, changes will not invalidate the addresses of pending changes.

Acme therefore avoids the initial transcript. Instead, changes are applied directly to the file, with an undo transcript recorded in a separate list. For example, when text is added to a window, it is added directly and a record of what to delete to restore the state is appended to the undo list. Each undo action and the file are marked with a sequence number; actions with the same sequence number are considered a unit to be undone together. The invariant state of the structure is that the last action in the undo list applies to the current state of the file, even if that action is one of a related set from, for example, a global substitute. (In Sam, a related set of actions needed to be undone simultaneously.) To undo an action, pop the last item on the undo list, apply it to the file, revert it, and append it to a second, redo list. To redo an action, do the identical operation with the lists interchanged. The expensive operations occur only when actually undoing; in normal editing the overhead is minor. For example, Acme reads files about seven times faster than Sam, partly because of this improvement and partly because of a cleaner implementation.

Acme uses a temporary file to hold the text, keeping in memory only the visible portion, and therefore can edit large files comfortably even on small-memory machines such as laptops.

Future

Acme is still under development. Some things are simply missing. For example, Acme should support non-textual graphics, but this is being deferred until it can be done using a new graphics model being developed for Plan 9. Also, it is undecided how Acme's style of interaction should best be extended to graphical applications. On a smaller scale, although the system feels smooth and comfortable, work continues to tune the heuristics and try new ideas for the user interface.

There need to be more programs that use Acme. Browsers for Usenet and AP News articles, the Oxford English Dictionary, and other such text sources exist, but more imaginative applications will be necessary to prove that Acme's approach is viable. One that has recently been started is an interface to the debugger Acid [Wint94], although it is still unclear what form it will ultimately take.

Acme shows that it is possible to make a user interface a stand-alone component of an interactive environment. By absorbing more of the interactive functionality than a simple window system, Acme off-loads much of the computation from its applications, which helps keep them small and consistent in their interface. Acme can afford to dedicate considerable effort to making that interface as good as possible; the result will benefit the entire system.

Acme is complete and useful enough to attract users. Its comfortable user interface, the ease with which it handles multiple tasks and programs in multiple directories, and its high level of integration make it addictive. Perhaps most telling, Acme shows that typescripts may not be the most productive interface to a time-sharing system.

Acknowledgements

Howard Trickey, Acme's first user, suffered buggy versions gracefully and made many helpful suggestions. Chris Fraser provided the necessary insight for the Acme editing commands.

References

- [ALEF] P. Winterbottom, "ALEF reference Manual", *Plan 9 Programmer's Manual*, AT&T Bell Laboratories, Murray Hill, NJ, 1992
- [Alle92] *Allegro Common Lisp user Guide*, Vol 2, Chapter 14, "The Emacs-Lisp Interface". March 1992
- [AT&T92] Plan 9 Programmer's manual, Murray Hill, New Jersey, 1992
- [Far89] Far too many people, XTERM(1), Massachusetts Institute of Technology, 1989
- [Gans93] Emden R. Gansner and John H. Reppy, "A Multi-threaded Higher-order User Interface Toolkit", in *Software Trends, Volume 1, User Interface Software*, Bass and Dewan (Eds.), John Wiley & Sons 1993, pp. 61-80
- [Lucid92] Richard Stallman and Lucid, Inc., *Lucid GNU EMACS Manual*, March 1992
- [Pike87] Rob Pike, "The Text Editor sam", *Softw. - Prac. and Exp.*, Nov 1987, Vol 17 #11, pp. 813-845
- [Pike88] Rob Pike, "Window Systems Should Be Transparent", *Comp. Sys.*, Summer 1988, Vol 1 #3, pp. 279-296
- [Pike89] Rob Pike, "A Concurrent Window System", *Comp. Sys.*, Spring 1989, Vol 2 #2, pp. 133-153
- [PPTTW93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom, *Op. Sys. Rev.*, Vol. 27, No. 2, April 1993, pp. 72-76
- [Pike91] Rob Pike, "8½, the Plan 9 Window System", *USENIX Summer Conf. Proc.*, Nashville, June, 1991, pp. 257-265
- [Pike92] Rob Pike, "A Minimalist Global User Interface", *Graphics Interface '92 Proc.*, Vancouver, 1992, pp. 282-293. An earlier version appeared under the same title in *USENIX Summer Conf. Proc.*, Nashville, June, 1991, pp. 267-279.
- [Pike93] Rob Pike and Ken Thompson, "Hello World or Καλημέρα κόσμε or こんにちは世界", *USENIX Winter Conf. Proc.*, San Diego, 1993, pp. 43-50.
- [Pres93] Dave Presotto and Phil Winterbottom, "The Organization of Networks in Plan 9", *Proc. Usenix Winter 1993*, pp. 271-287, San Diego, CA.
- [Reis91] Martin Reiser, *The Oberon System*, Addison Wesley, New York, 1991
- [Reppy93] John H. Reppy, "CML: A higher-order concurrent language", *Proc. SIGPLAN'91 Conf. on Programming, Lang. Design and Impl.*, June, 1991, pp. 293-305
- [Sche86] Robert W. Scheifler and Jim Gettys, "The X Window System", *ACM Trans. on Graph.*, Vol 5 #2, pp. 79-109
- [Stal93] Richard Stallman, *Gnu Emacs Manual*, 9th edition, Emacs version 19.19, MIT
- [Swei86] Daniel Sweinhart, Polle Zellweger, Richard Beach, and Robert Hagmann, "A Structural View of the Cedar Programming Environment", *ACM Trans. Prog. Lang. and Sys.*, Vol. 8, No. 4, pp. 419-490, Oct. 1986.
- [Wint94], Philip Winterbottom, "ACID: A Debugger based on a Language", *USENIX Winter Conf. Proc.*, San Francisco, CA, 1993.
- [Wirt89] N. Wirth and J. Gutknecht, "The Oberon System", *Softw. - Prac. and Exp.*, Sep 1989, Vol 19 #9, pp 857-894

Biography

Rob Pike, well known for his appearances on "Late Night with David Letterman", is also a Member of Technical Staff at AT&T Bell Laboratories in Murray Hill, New Jersey, where he has been since 1980, the same year he won the Olympic silver medal in Archery. In 1981 he wrote the first bitmap window system for Unix systems, and has since written ten more. With Bart Locanthi he designed the Blit terminal; with Brian Kernighan he wrote *The Unix Programming Environment*. A shuttle mission nearly launched a gamma-ray telescope he designed. He is a Canadian citizen and has never written a program that uses cursor addressing.

File System Design for an NFS File Server Appliance

Dave Hitz

James Lau

Michael Malcolm

Network Appliance Corporation

Abstract

Network Appliance Corporation recently began shipping a new kind of network server called an NFS file server appliance, which is a dedicated server whose sole function is to provide NFS file service. The file system requirements for an NFS appliance are different from those for a general-purpose UNIX system, both because an NFS appliance must be optimized for network file access and because an appliance must be easy to use.

This paper describes WAFL (Write Anywhere File Layout), which is a file system designed specifically to work in an NFS appliance. The primary focus is on the algorithms and data structures that WAFL uses to implement Snapshots™, which are read-only clones of the active file system. WAFL uses a copy-on-write technique to minimize the disk space that Snapshots consume. This paper also describes how WAFL uses Snapshots to eliminate the need for file system consistency checking after an unclean shutdown.

1. Introduction

An appliance is a device designed to perform a particular function. A recent trend in networking has been to provide common services using appliances instead of general-purpose computers. For instance, special-purpose routers from companies like Cisco and Wellfleet have almost entirely replaced general-purpose computers for packet routing, even though general purpose computers originally handled all routing. Other examples of network appliances include network terminal concentrators, network FAX servers, and network printers.

A new type of network appliance is the NFS file server appliance. The requirements for a file system operating in an NFS appliance are different from those for a general purpose file system: NFS access patterns are different than local access patterns, and the special-purpose nature of an appliance also affects the design.

WAFL (Write Anywhere File Layout) is the file system used in Network Appliance Corporation's FAServer™ NFS appliance. WAFL was designed to meet four primary requirements:

- (1) It should provide fast NFS service.
- (2) It should support large file systems (tens of GB) that grow dynamically as disks are added.
- (3) It should provide high performance while supporting RAID (Redundant Array of Inexpensive Disks).
- (4) It should restart quickly, even after an unclean shutdown due to power failure or system crash.

The requirement for fast NFS service is obvious, given WAFL's intended use in an NFS appliance. Support for large file systems simplifies system administration by allowing all disk space to belong to a single large partition. Large file systems make RAID desirable because the probability of disk failure increases with the number of disks. Large file systems require special techniques for fast restart because the file system consistency checks for normal UNIX file systems become unacceptably slow as file systems grow.

NFS and RAID both strain write performance: NFS because servers must store data safely before replying to NFS requests, and RAID because of the read-modify-write sequence it uses to maintain parity [Patterson88]. This led us to use non-volatile RAM to reduce NFS response time and a write-anywhere design that allows WAFL to write to disk locations that minimize RAID's write performance penalty. The write-anywhere design enables Snapshots, which in turn eliminate the requirement for time-consuming consistency checks after power loss or system failure.

2. Introduction To Snapshots

WAFL's primary distinguishing characteristic is Snapshots, which are read-only copies of the entire file system. WAFL creates and deletes Snapshots automatically at prescheduled times, and it keeps up to 20 Snapshots on-line at once to provide easy access to old versions of files.

Snapshots use a copy-on-write technique to avoid duplicating disk blocks that are the same in a Snapshot as in the active file system. Only when blocks in the active file system are modified or removed do Snapshots containing those blocks begin to consume disk space.

Users can access Snapshots through NFS to recover files that they have accidentally changed or removed, and system administrators can use Snapshots to create backups safely from a running system. In addition, WAFL uses Snapshots internally so that it can restart quickly even after an unclean system shutdown.

2.1. User Access to Snapshots

Every directory in the file system contains a hidden sub-directory named `.snapshot` that allows users to access the contents of Snapshots over NFS. Suppose that a user has accidentally removed a file named `todo` and wants to recover it. The following example shows how to list all the versions of `todo` saved in Snapshots:

```
spike% ls -lut .snapshot/*/todo
-rw-r--r-- 1 hitz      52880 Oct 15 00:00 .snapshot/nightly.0/todo
-rw-r--r-- 1 hitz      52880 Oct 14 19:00 .snapshot/hourly.0/todo
-rw-r--r-- 1 hitz      52829 Oct 14 15:00 .snapshot/hourly.1/todo
...
-rw-r--r-- 1 hitz      55059 Oct 10 00:00 .snapshot/nightly.4/todo
-rw-r--r-- 1 hitz      55059 Oct  9 00:00 .snapshot/nightly.5/todo
```

With the `-u` option, `ls` shows `todo`'s access time, which is set to the time when the Snapshot containing it was created. The user can recover the most recent version of `todo` by copying it back into the current directory:

```
spike% cp .snapshot/hourly.0/todo .
```

The `.snapshot` directories are "hidden" in the sense that they do not show up in directory listings. If `.snapshot` were visible, commands like `find` would report many more files than expected, and commands like `rm -rf` would fail because files in Snapshots are read-only and cannot be removed.

2.2. Snapshot Administration

The FAServer has commands to let system administrators create and delete Snapshots, but it creates and deletes most Snapshots automatically. By default, the FAServer creates four hourly Snapshots at various times during the day, and a nightly Snapshot every night at midnight. It keeps the hourly

Snapshots for two days, and the nightly Snapshots for a week. It also creates a weekly Snapshot at midnight on Sunday, which it keeps for two weeks.

For file systems that change quickly, this schedule may consume too much disk space, and Snapshots may need to be deleted sooner. A Snapshot is useful even if it is kept for just a few hours, because users usually notice immediately when they have removed an important file. For file systems that change slowly, it may make sense to keep Snapshots on-line for longer. In typical environments, keeping Snapshots for one week consumes 10 to 20 percent of disk space.

3. WAFL Implementation

3.1. Overview

WAFL is a UNIX compatible file system optimized for network file access. In many ways WAFL is similar to other UNIX file systems such as the Berkeley Fast File System (FFS) [McKusick84] and TransArc's Episode file system [Chutani92]. WAFL is a block-based file system that uses inodes to describe files. It uses 4 KB blocks with no fragments.

Each WAFL inode contains 16 block pointers to indicate which blocks belong to the file. Unlike FFS, all the block pointers in a WAFL inode refer to blocks at the same level. Thus, inodes for files smaller than 64 KB use the 16 block pointers to point to data blocks. Inodes for files smaller than 64 MB point to indirect blocks which point to actual file data. Inodes for larger files point to doubly indirect blocks. For very small files, data is stored in the inode itself in place of the block pointers.

3.2. Meta-Data Lives in Files

Like Episode, WAFL stores meta-data in files. WAFL's three meta-data files are the inode file, which contains the inodes for the file system, the block-map file, which identifies free blocks, and the inode-map file, which identifies free inodes. The term "map" is used instead of "bit map" because these files use more than one bit for each entry. The block-map file's format is described in detail below.

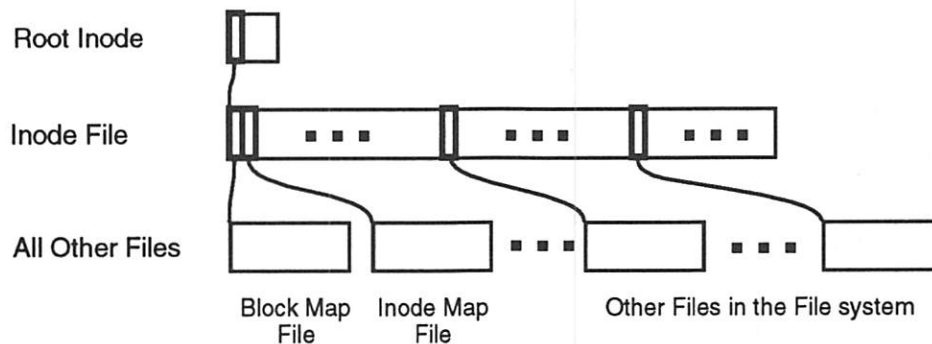


Figure 1: The WAFL file system is a tree of blocks with the root inode, which describes the inode file, at the top, and meta-data files and regular files underneath.

Keeping meta-data in files allows WAFL to write meta-data blocks anywhere on disk. This is the origin of the name WAFL, which stands for Write Anywhere File Layout. The write-anywhere design allows WAFL to operate efficiently with RAID by scheduling multiple writes to the same RAID stripe whenever possible to avoid the 4-to-1 write penalty that RAID incurs when it updates just one block in a stripe.

Keeping meta-data in files makes it easy to increase the size of the file system on the fly. When a new disk is added, the FAServer automatically increases the sizes of the meta-data files. The system administrator can increase the number of inodes in the file system manually if the default is too small.

Finally, the write-anywhere design enables the copy-on-write technique used by Snapshots. For Snapshots to work, WAFL must be able to write all new data, including meta-data, to new locations on

disk, instead of overwriting the old data. If WAFL stored meta-data at fixed locations on disk, this would not be possible.

3.3. Tree of Blocks

A WAFL file system is best thought of as a tree of blocks. At the root of the tree is the root inode, as shown in Figure 1. The root inode is a special inode that describes the inode file. The inode file contains the inodes that describe the rest of the files in the file system, including the block-map and inode-map files. The leaves of the tree are the data blocks of all the files.

Figure 2 is a more detailed version of Figure 1. It shows that files are made up of individual blocks and that large files have additional layers of indirection between the inode and the actual data blocks. In order for WAFL to boot, it must be able to find the root of this tree, so the one exception to WAFL's write-anywhere rule is that the block containing the root inode must live at a fixed location on disk where WAFL can find it.

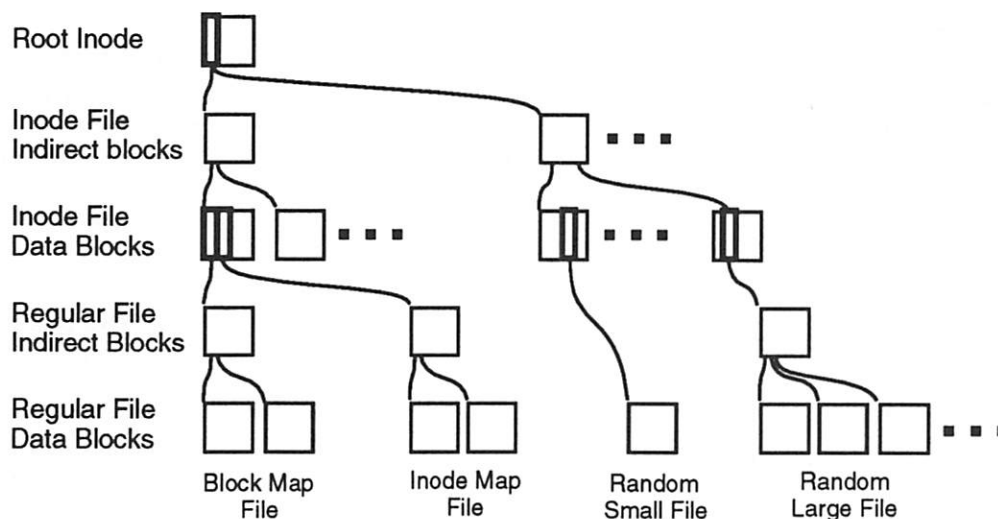


Figure 2: A more detailed view of WAFL's tree of blocks.

3.4. Snapshots

Understanding that the WAFL file system is a tree of blocks rooted by the root inode is the key to understanding Snapshots. To create a virtual copy of this tree of blocks, WAFL simply duplicates the root inode. Figure 3 shows how this works.

Figure 3(a) is a simplified diagram of the file system in Figure 2 that leaves out internal nodes in the tree, such as inodes and indirect blocks.

Figure 3(b) shows how WAFL creates a new Snapshot by making a duplicate copy of the root inode. This duplicate inode becomes the root of a tree of blocks representing the Snapshot, just as the root inode represents the active file system. When the Snapshot inode is created, it points to exactly the same disk blocks as the root inode, so a brand new Snapshot consumes no disk space except for the Snapshot inode itself.

Figure 3(c) shows what happens when a user modifies data block D. WAFL writes the new data to block D' on disk, and changes the active file system to point to the new block. The Snapshot still references the original block D which is unmodified on disk. Over time, as files in the active file system are modified or deleted, the Snapshot references more and more blocks that are no longer used in the active file system. The rate at which files change determines how long Snapshots can be kept on line before they consume an unacceptable amount of disk space.

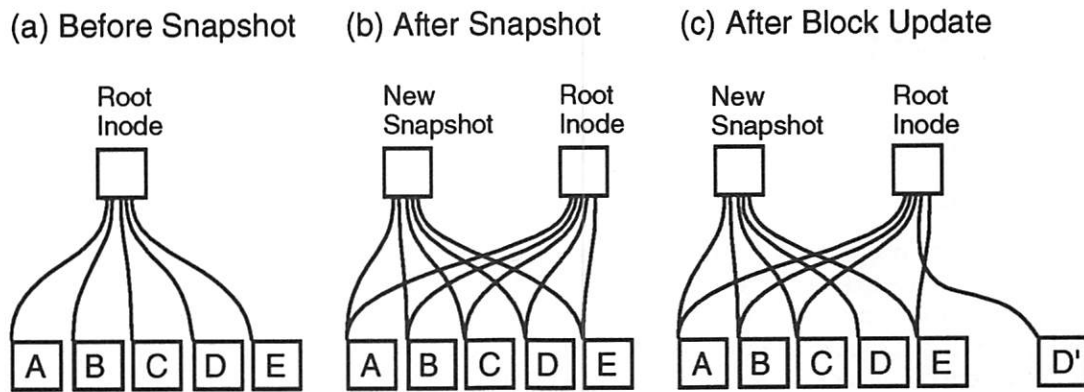


Figure 3: WAFL creates a Snapshot by duplicating the root inode which describes the inode file. WAFL avoids changing blocks in a Snapshot by writing new data to new locations on disk.

It is interesting to compare WAFL's Snapshots with Episode's fileset clones. Instead of duplicating the root inode, Episode creates a clone by copying the entire inode file and all the indirect blocks in the file system. This generates considerable disk I/O and consumes a lot of disk space. For instance, a 10 GB file system with one inode for every 4 KB of disk space would have 320 MB of inodes. In such a file system, creating a Snapshot by duplicating the inodes would generate 320 MB of disk I/O and consume 320 MB of disk space. Creating 10 such Snapshots would consume almost one-third of the file system's space even before any data blocks were modified.

By duplicating just the root inode, WAFL creates Snapshots very quickly and with very little disk I/O. Snapshot performance is important because WAFL creates a Snapshot every few seconds to allow quick recovery after unclean system shutdowns.

Figure 4 shows the transition from Figure 3(b) to 3(c) in more detail. When a disk block is modified, and its contents are written to a new location, the block's parent must be modified to reflect the new location. The parent's parent, in turn, must also be written to a new location, and so on up to the root of the tree.

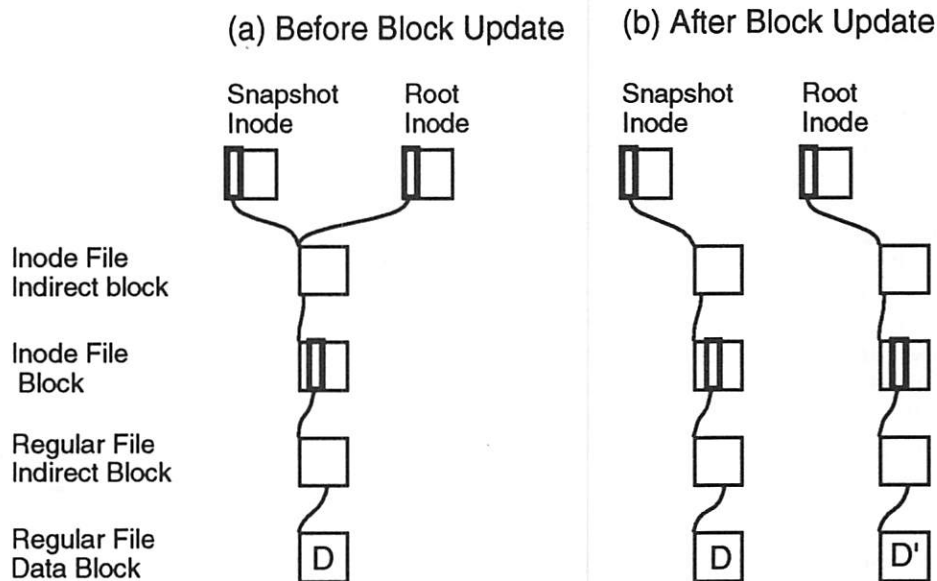


Figure 4: To write a block to a new location, the pointers in the block's ancestors must be updated, which requires them to be written to new locations as well.

WAFL would be very inefficient if it wrote this many blocks for each NFS write request. Instead, WAFL gathers up many hundreds of NFS requests before scheduling a write episode. During a write episode, WAFL allocates disk space for all the dirty data in the cache and schedules the required disk I/O. As a result, commonly modified blocks, such as indirect blocks and blocks in the inode file, are written only once per write episode instead of once per NFS request.

3.5. File System Consistency and Non-Volatile RAM

WAFL avoids the need for file system consistency checking after an unclean shutdown by creating a special Snapshot called a *consistency point* every few seconds. Unlike other Snapshots, a consistency point has no name, and it is not accessible through NFS. Like all Snapshots, a consistency point is a completely self consistent image of the entire file system. When WAFL restarts, it simply reverts to the most recent consistency point. This allows a FAServer to reboot in about a minute even with 20 GB or more of data in its single partition.

Between consistency points, WAFL does write data to disk, but it writes only to blocks that are not in use, so the tree of blocks representing the most recent consistency point remains completely unchanged. WAFL processes hundreds or thousands of NFS requests between consistency points, so the on-disk image of the file system remains the same for many seconds until WAFL writes a new consistency point, at which time the on-disk image advances atomically to a new state that reflects the changes made by the new requests. Although this technique is unusual for a UNIX file system, it is well known for databases. See, for instance, [Astrahan76] which describes the shadow paging technique used in System R. Even in databases it is unusual to write as many operations at one time as WAFL does in its consistency points.

WAFL uses non-volatile RAM (NVRAM) to keep a log of NFS requests it has processed since the last consistency point. (NVRAM is special memory with batteries that allow it to store data even when system power is off.) After an unclean shutdown, WAFL replays any requests in the log to prevent them from being lost. When a FAServer shuts down normally, it creates one last consistency point after suspending NFS service. Thus, on a clean shutdown the NVRAM doesn't contain any unprocessed NFS requests, and it is turned off to increase its battery life.

WAFL actually divides the NVRAM into two separate logs. When one log gets full, WAFL switches to the other log and starts writing a consistency point to store the changes from the first log safely on disk. WAFL schedules a consistency point every 10 seconds, even if the log is not full, to prevent the on-disk image of the file system from getting too far out of date.

Logging NFS requests to NVRAM has several advantages over the more common technique of using NVRAM to cache writes at the disk driver layer. Lyon and Sandberg describe the NVRAM write cache technique, which Legato's Prestoserve™ NFS accelerator uses [Lyon89].

Processing an NFS request and caching the resulting disk writes generally takes much more NVRAM than simply logging the information required to replay the request. For instance, to move a file from one directory to another, the file system must update the contents and inodes of both the source and target directories. In FFS, where blocks are 8 KB each, this uses 32 KB of cache space. WAFL uses about 150 bytes to log the information needed to replay a rename operation. Rename—with its factor of 200 difference in NVRAM usage—is an extreme case, but even for a simple 8 KB write, caching disk blocks will consume 8 KB for the data, 8 KB for the inode update, and—for large files—another 8 KB for the indirect block. WAFL logs just the 8 KB of data along with about 120 bytes of header information. With a typical mix of NFS operations, WAFL can store more than 1000 operations per megabyte of NVRAM.

Using NVRAM as a cache of unwritten disk blocks turns it into an integral part of the disk subsystem. An NVRAM failure can corrupt the file system in ways that `fsck` cannot detect or repair. If something goes wrong with WAFL's NVRAM, WAFL may lose a few NFS requests, but the on-disk image of the file system remains completely self consistent. This is important because NVRAM is reliable, but not as reliable as a RAID disk array.

A final advantage of logging NFS requests is that it improves NFS response times. To reply to an NFS request, a file system without any NVRAM must update its in-memory data structures, allocate disk space for new data, and wait for all modified data to reach disk. A file system with an NVRAM write cache does all the same steps, except that it copies modified data into NVRAM instead of waiting for the data to reach disk. WAFL can reply to NFS an request much more quickly because it need only update its in-memory data structures and log the request. It does not allocate disk space for new data or copy modified data to NVRAM.

3.6. Write Allocation

Write performance is especially important for network file servers. Ousterhout observed that as read caches get larger at both the client and server, writes begin to dominate the I/O subsystem [Ousterhout89]. This effect is especially pronounced with NFS which allows very little client-side write caching. The result is that the disks on an NFS server may have 5 times as many write operations as reads.

WAFL's design was motivated largely by a desire to maximize the flexibility of its write allocation policies. This flexibility takes three forms:

- (1) WAFL can write any file system block (except the one containing the root inode) to any location on disk.

In FFS, meta-data, such as inodes and bit maps, is kept in fixed locations on disk. This prevents FFS from optimizing writes by, for example, putting both the data for a newly updated file and its inode right next to each other on disk. Since WAFL can write meta-data anywhere on disk, it can optimize writes more creatively.

- (2) WAFL can write blocks to disk in any order.

FFS writes blocks to disk in a carefully determined order so that `fsck(8)` can restore file system consistency after an unclean shutdown. WAFL can write blocks in any order because the on-disk image of the file system changes only when WAFL writes a consistency point. The one constraint is that WAFL must write all the blocks in a new consistency point before it writes the root inode for the consistency point.

- (3) WAFL can allocate disk space for many NFS operations at once in a single write episode.

FFS allocates disk space as it processes each NFS request. WAFL gathers up hundreds of NFS requests before scheduling a consistency point, at which time it allocates blocks for all requests in the consistency point at once. Deferring write allocation improves the latency of NFS operations by removing disk allocation from the processing path of the reply, and it avoids wasting time allocating space for blocks that are removed before they reach disk.

These features gives WAFL extraordinary flexibility in its write allocation policies. The ability to schedule writes for many requests at once enables more intelligent allocation policies, and the fact that blocks can be written to any location and in any order allows a wide variety of strategies. It is easy to try new block allocation strategies without any change to WAFL's on-disk data structures.

The details of WAFL's write allocation policies are outside the scope of this paper. In short, WAFL improves RAID performance by writing to multiple blocks in the same stripe; WAFL reduces seek time by writing blocks to locations that are near each other on disk; and WAFL reduces head-contention when reading large files by placing sequential blocks in a file on a single disk in the RAID array. Optimizing write allocation is difficult because these goals often conflict.

4. Snapshot Data Structures And Algorithms

4.1. The Block-Map File

Most file systems keep track of free blocks using a bit map with one bit per disk block. If the bit is set, then the block is in use. This technique does not work for WAFL because many snapshots can reference a

block at the same time. WAFL's block-map file contains a 32-bit entry for each 4 KB disk block. Bit 0 is set if the active file system references the block, bit 1 is set if the first Snapshot references the block, and so on. A block is in use if any of the bits in its block-map entry are set.

Figure 5 shows the life cycle of a typical block-map entry. At time $t1$, the block-map entry is completely clear, indicating that the block is available. At time $t2$, WAFL allocates the block and stores file data in it. When Snapshots are created, at times $t3$ and $t4$, WAFL copies the active file system bit into the bit indicating membership in the Snapshot. The block is deleted from the active file system at time $t5$. This can occur either because the file containing the block is removed, or because the contents of the block are updated and the new contents are written to a new location on disk. The block can't be reused, however, until no Snapshot references it. In Figure 5, this occurs at time $t8$ after both Snapshots that reference the block have been removed.

Time	Block-Map Entry	Description
$t1$	0 0 0 0 0 0 0 0	Block is unused.
$t2$	0 0 0 0 0 0 0 1	Block is allocated for active FS
$t3$	0 0 0 0 0 0 1 1	Snapshot #1 is created
$t4$	0 0 0 0 0 1 1 1	Snapshot #2 is created
$t5$	0 0 0 0 0 1 1 0	Block is deleted from active FS
$t6$	0 0 0 0 0 1 1 0	Snapshot #3 is created
$t7$	0 0 0 0 0 1 0 0	Snapshot #1 is deleted
$t8$	0 0 0 0 0 0 0 0	Snapshot #2 is deleted; block is unused

- bit 0: set for active file system
- bit 1: set for Snapshot #1
- bit 2: set for Snapshot #2
- bit 3: set for Snapshot #3

Figure 5: The life cycle of a block-map file entry.

4.2. Creating a Snapshot

The challenge in writing a Snapshot to disk is to avoid locking out incoming NFS requests. The problem is that new NFS requests may need to change cached data that is part of the Snapshot and which must remain unchanged until it reaches disk. An easy way to create a Snapshot would be to suspend NFS processing, write the Snapshot, and then resume NFS processing. However, writing a Snapshot can take over a second, which is too long for an NFS server to stop responding. Remember that WAFL creates a consistency point Snapshot at least every 10 seconds, so performance is critical.

WAFL's technique for keeping Snapshot data self consistent is to mark all the dirty data in the cache as "IN_SNAPSHOT". The rule during Snapshot creation is that data marked IN_SNAPSHOT must not be modified, and data not marked IN_SNAPSHOT must not be flushed to disk. NFS requests can read all file system data, and they can modify data that isn't IN_SNAPSHOT, but processing for requests that need to modify IN_SNAPSHOT data must be deferred.

To avoid locking out NFS requests, WAFL must flush IN_SNAPSHOT data as quickly as possible. To do this, WAFL performs the following steps:

- (1) Allocate disk space for all files with IN_SNAPSHOT blocks. WAFL caches inode data in two places: in a special cache of in-core inodes, and in disk buffers belonging to the inode file. When it finishes write allocating a file, WAFL copies the newly updated inode information from the inode cache into the appropriate inode file disk buffer, and clears the IN_SNAPSHOT bit on the in-core inode. When this step is complete no inodes for regular files are marked IN_SNAPSHOT, and most NFS operations can continue without blocking. Fortunately, this step can be done very quickly because it requires no disk I/O.

- (2) Update the block-map file. For each block-map entry, WAFL copies the bit for the active file system to the bit for the new Snapshot.
- (3) Write all IN_SNAPSHOT disk buffers in cache to their newly-allocated locations on disk. As soon as a particular buffer is flushed, WAFL restarts any NFS requests waiting to modify it.
- (4) Duplicate the root inode to create an inode that represents the new Snapshot, and turn the root inode's IN_SNAPSHOT bit off. The new Snapshot inode must not reach disk until after all other blocks in the Snapshot have been written. If this rule were not followed, an unexpected system shutdown could leave the Snapshot in an inconsistent state.

Once the new Snapshot inode has been written, no more IN_SNAPSHOT data exists in cache, and any NFS requests that are still suspended can be processed. Under normal loads, WAFL performs these four steps in less than a second. Step (1) can generally be done in just a few hundredths of a second, and once WAFL completes it, very few NFS operations need to be delayed.

Deleting a Snapshot is trivial. WAFL simply zeros the root inode representing the Snapshot and clears the bit representing the Snapshot in each block-map entry.

5. Performance

It is difficult to compare WAFL's performance to other file systems directly. Since WAFL runs only in an NFS appliance, it can be benchmarked against other file systems only in the context of NFS. The best NFS benchmark available today is the SPEC SFS (System File Server) benchmark, also known as LADDIS. The name LADDIS stands for the group of companies that originally developed the benchmark: Legato, Aupex, Digital, Data General, Interphase, and Sun.

LADDIS tests NFS performance by measuring a server's response time at various throughput levels. Servers typically handle requests most quickly at low load levels; as the load increases, so does the response time. Figure 6 compares the FAServer's LADDIS performance with that of other well-known NFS servers.

Using a system level benchmark, such as LADDIS, to compare file system performance can be misleading. One might argue, for instance, that Figure 6 underestimates WAFL's performance because the FAServer cluster has only 8 file systems, while the other servers all have dozens. On a per file system basis, WAFL outperforms the other file systems by almost eight to one. Furthermore, the FAServer uses RAID (which typically degrades file system performance substantially for the small request sizes characteristic of NFS), whereas the other servers do not use RAID.

On the other hand, one might argue that the benchmark overestimates WAFL's performance, because the entire FAServer is designed specifically for NFS, and much of its performance comes from NFS-specific tuning of the whole system—not just to WAFL.

Given WAFL's special purpose nature, there is probably no fair way to compare its performance to general purpose file systems, but it clearly satisfies the design goal of performing well with NFS and RAID.

6. Conclusion

WAFL was developed, and has become stable, surprisingly quickly for a new file system. It has been in use as a production file system for over a year, and we know of no case where it has lost user data.

We attribute this stability in part to WAFL's use of consistency points. Processing file system requests is simple because WAFL updates only in-memory data structures and the NVRAM log. Consistency points eliminate ordering constraints for disk writes, which are a significant source of bugs in most file systems. The code that writes consistency points is concentrated in a single file, it interacts little with the rest of WAFL, and it executes relatively infrequently.

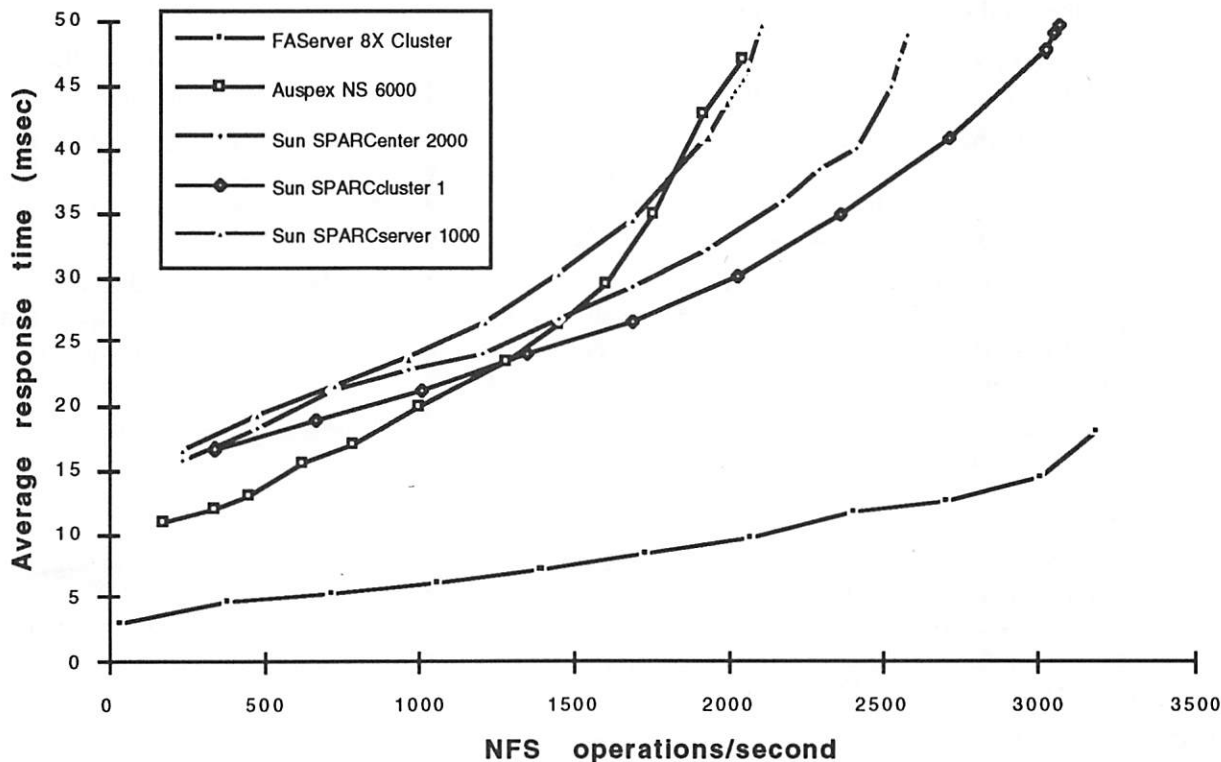


Figure 6: Graph of SPECnfs_A93 operations per second. (For clusters, the graph shows SPECnfs_A93 cluster operations per second.)

More importantly, we believe that it is much easier to develop high quality, high performance system software for an appliance than for a general purpose operating system. Compared to a general purpose file system, WAFL handles a very regular and simple set of requests. A general purpose file system receives requests from thousands of different applications with a wide variety of different access patterns, and new applications are added frequently. By contrast, WAFL receives requests only from the NFS client code of other systems. There are few NFS client implementations, and new implementations are rare. Of course, applications are the ultimate source of NFS requests, but the NFS client code converts file system requests into a regular pattern of network requests, and it filters out error cases before they reach the server. The small number of operations that WAFL supports makes it possible to define and test the entire range of inputs that it is expected to handle.

These advantages apply to any appliance, not just to file server appliances. A network appliance only makes sense for protocols that are well defined and widely used, but for such protocols, an appliance can provide important advantages over a general purpose computer.

References

- [Astrahan76] M. Astrahan, M. Blasgen, K. Chamberlain, K. Eswaran, J. Gravy, P. Griffiths, W. King, I. Traiger, B. Wade and V. Watson.
System R: Relational Approach to Database Management.
ACM Transactions on Database Systems 1, 2 (1976), pp. 97-137.
- [Chutani92] Sailesh Chutani, et. al.
The Episode File System.
Proceedings of the Winter 1992 USENIX Conference, pp. 43-60, San Francisco, CA, January 1992.
- [Hitz93] Dave Hitz.
An NFS File Server Appliance.
Network Appliance Corporation, 2901 Tasman Drive, Suite 208, Santa Clara, CA 95054
- [Lyon89] Bob Lyon and Russel Sandberg.
Breaking Through the NFS Performance Barrier.
SunTech Journal 2(4): 21-27, Autumn 1989.
- [McKusick84] Marshall K. McKusick.
A Fast File System for UNIX.
ACM Transactions on Computer Systems 2(3): 181-97, August 1984.
- [Ousterhout89] John Ousterhout and Fred Douglass
Beating the I/O Bottleneck: A Case for Log-Structured File Systems.
ACM SIGOPS, 23, January 1989.
- [Patterson88] D. Patterson, G. Gibson, and R. Katz.
A Case for Redundant Arrays of Inexpensive Disks (RAID).
ACM SIGMOD 88, Chicago, June 1988, pp. 109-116.
- [Sandberg85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon.
Design and Implementation of the Sun Network File System.
Proceedings of the Summer 1985 USENIX Conference, pp. 119-30, Portland, OR, June 1985.

Biographies

Dave Hitz is a co-founder and system architect at Network Appliance Corporation, which builds NFS file server appliances. At Network Appliance Dave has focused on designing and implementing the Network Appliance file system, and on the overall design of the Network Appliance file server. He also worked at Auspex Systems in the file system group, and at MIPS in the System V kernel group. Other jobs and hobbies have included herding, castrating, and slaughtering cattle, pen-based computer programming, and typing names onto Blue Shield Insurance cards. After dropping out of high school, he attended George Washington University, Swarthmore College, Deep Springs College, and finally Princeton University where he received his computer science BSE in 1986.

James Lau is a co-founder and director of engineering at Network Appliance Corporation. Before that, James spent three years at Auspex Systems, most recently as director of software engineering. He was instrumental in defining product requirements and the high level architecture of Auspex's high performance NFS file server. Before joining Auspex, James spent five years at Bridge Communications where he implemented a variety of protocols in XNS, TCP/IP, Ethernet, X25, and HDLC. He spent the last year at Bridge as the group manager of PC products. James received his masters degree in computer engineering from Stanford and bachelors degrees in computer science and applied mathematics from U.C. Berkeley.

Michael Malcolm is co-founder, President and CEO of Network Appliance Corporation. Previously, he ran a successful management consulting practice with clients focused on distributed computing, networking, and file storage technology. He was founder and CEO of Waterloo Microsystems, a Canadian developer of network operating system software. In the past, he was an Associate Professor of Computer Science at University of Waterloo where he taught hundreds of students how to program real-time systems to control electric model trains. His research spanned the areas of network operating systems, portable operating systems, interprocess communication, compiler design, and numerical mathematics. He led the development of two major operating systems: Thoth, and Waterloo Port. He received a B.S. in Mechanical Engineering from University of Denver in 1966, and a Ph.D. in Computer Science from Stanford University in 1973.

The authors can be reached at hitz@netapp.com, jlau@netapp.com, and malcolm@netapp.com.

Improving the Write Performance of an NFS Server

Chet Juszczak - Digital Equipment Corporation

ABSTRACT

The Network File System (NFS) utilizes a stateless protocol between clients and servers; the major advantage of this statelessness is that NFS crash recovery is very easy. However, the protocol requires that data modification operations such as *write* be fully committed to stable storage before replying to the client. The cost of this is significant in terms of response latency and server CPU and I/O loading. This paper describes a *write gathering* technique that exploits the fact that there are often several write requests for the same file presented to the server at about the same time. With this technique the data portions of these writes are combined and a single metadata update is done that applies to them all. No replies are sent to the client until after this metadata update has been fully committed, thus the NFS crash recovery design is not violated. This technique can be used in most NFS server implementations and requires no client modifications.

1. Introduction

The Network File System (NFS) remains a de facto standard in the UNIX industry. NFS utilizes a stateless protocol between clients and servers. The major advantage of this statelessness is that NFS crash recovery is very easy. Neither client nor server must detect the other's crashes. Since a server has no state information to maintain, there is nothing for it to throw away after a client crashes. Likewise, there is no state information to re-build when the server returns after a crash. However, this protocol requires that data modification operations, e.g. *write*, be fully committed to stable storage before replying to the client [SAND85].

The write operation is usually the most costly of all NFS server operations (see *NFS Writes*, below). A heavy write load typically yields poorer server performance than loads of other types. It is not unusual, e.g., to see the write operation portion (15%) of the SPEC SFS 1.0 NFS server benchmark (SPEC/LADDIS) account for two thirds or more of the total latency and contribute more than its share to server CPU and I/O loading [WITT93]. Also, the large latencies of NFS write operations can lead to more serious problems on the server when it is faced with retransmissions from impatient clients [JUSZ89].

This paper describes the implementation of a *write gathering* technique that exploits the fact that there are often several write requests for the same file presented to the server at about the same time. With this technique the data portions of these writes are combined and a single metadata update is done that applies to them all. No replies are sent to the client until after this metadata update has been fully committed, thus the NFS crash recovery design is not violated. This technique can be used in most NFS server implementations. It requires no client modifications; it exploits behavior that is common among existing workstation clients. The implementation of this technique has resulted in a significant increase in server write bandwidth and an increase in overall server capacity and responsiveness (see *Results*).

2. Related Work

I do not claim to have originated the notion of write gathering as a performance optimization for NFS writes. Prior to 1990, while at Epoch Systems, Inc., Dave Noveck did some similar work within the filesystem below the NFS server layer [BROW90]. Dave is now at the Open Software Foundation, Cambridge, MA. Drew Perkins, while at Interstream, did an NFS server layer implementation for a SunOS after-market product; a demo of this product at the 1991 NFS Connectathon event got me interested in the problem. Drew is now at Fore Systems, Inc., Pittsburgh, PA. SUN has an NFS server layer implementation in current versions of Solaris that may make its way into a future *reference port* of NFS; it is rumored that they chose from one of several in house implementations. I suspect other NFS server vendors also have implementations. Suresh Sivaprakasam describes an implementation for SunOS that clusters NFS writes in [SIVA93]; I make some comparisons to my implementation later (see *Write Gathering*).

I do claim to have an original implementation that has proven to be useful. I found no discussion of this topic in the literature when I did the work in 1991/92; to date, I can find only [SIVA93]. Thanks to some nudging by Jeff Mogul, I am sharing my implementation details via this paper in the interest of sparking some discussion. Hopefully the level of NFS service provided by the vendor community will benefit.

3. Goals

My goal in originating this work in 1991 was to increase NFS write speed from the perspective of a single client writing a large file, and to do so without consuming an inordinate amount of server capacity. We were in the process of modifying our local filesystem to cluster reads and writes for higher disk file throughputs (similar to [MCVO91]). I wanted remote NFS clients to achieve throughputs similar to those of local processes modulo network limitations.

4. Background

This section contains background information on various NFS topics; it is provided to help understand the environment in which write gathering operates. References to "typical" NFS clients and servers refer to implementations of NFS derived from the 4.3BSD based kernel implementation available from Sun Microsystems, Inc. Similarly, the term *reference port* will refer to this implementation. My work was done with version 4.2 of the reference port. The following characterizations should largely apply to other NFS implementations as well.

4.1. NFS Clients

NFS client systems range in size from single threaded (dumb) PCs to small workstations to large multi-processor timesharing systems with hundreds of users.

Typical NFS clients (and servers) communicate via UDP messages with an effective maximum size (excluding protocol headers) of 8K. A typical client will retransmit a request if it has not received a response from the server for that request within an interval of time that defaults to a starting value of 1.1 seconds. The retransmission interval is dynamically adjusted based on past server performance. Server write performance is an important part of the client backoff algorithm. Since the write operation is typically the most expensive for the server to perform, and with the highest latency, write performance is used as an indicator of server performance for heavyweight operation types. Poor write performance will affect client behavior with respect to other types of requests. (The other two indicators are performance of read (middleweight) and lookup (lightweight) operations.)

A client system can have multiple outstanding read and/or write requests. A client process blocks whenever a read or write request cannot be satisfied locally and must be processed by the server. When it blocks, another process can run; that process may also generate a read or write request. A single process can have multiple outstanding read and/or write requests if the client system is running NFS block I/O (*biod(8)* or *nfsiod(8)*) daemons, referred to simply as *biods* from here on. Biods perform client read-ahead and write-behind functions asynchronously, allowing the client process to continue execution in parallel with client/server communication.

Let's assume a typical workstation class client system that is running biods, and a client process, C, that is generating write requests. When C generates a write that "needs to go to the wire" (a client kernel decision, typically decided by the application writing to the end of an 8K cache block), the task of communicating the write request is handed off to a biod and C continues. If no biod is available for hand-off, then C will block until *that*

particular request has received a response. This is independent of responses received for earlier writes handed off to bioses. The blocking of C provides a simple client/server flow control.

Most NFS clients impose a *sync on close* semantic where the `close(2)` call blocks until all outstanding writes have received responses. Mostly this is to capture an `ENOSPC` server response to asynchronous write requests and communicate it to the client application process.

4.2. NFS Servers

NFS server systems range in size from workstation class systems with several disks to large multi-processor systems with disk farms.

A typical NFS server system simply waits for work to appear on an incoming request queue. This queue is the socket buffer allocated for the NFS socket. Incoming requests are converted into a form understandable by the local filesystem routines that actually perform the work of getting data to/from a disk. The incoming request queue is typically of fixed size. If the queue fills (requests coming in faster than they can be processed) then some incoming requests may be lost and client backoff/retransmission comes into play. The server depends upon its clients to attenuate their request loads as it becomes heavily loaded (i.e. the aggregate load is coming in faster than can be processed). Write requests are typically large and time consuming; processing them more quickly and efficiently can help keep a server ahead of its clients.

The amount of work that a server can perform is called server bandwidth or capacity. It is usually limited by exhaustion of one of the following three:

- CPU capacity,
- network interface capacity, or
- disk subsystem capacity.

Server capacity is sometimes measured in a general manner, e.g. NFS operations/second over some sort of operation mix (e.g. SPEC/LADDIS), and sometimes specifically, e.g. read or write speed in Kbytes/second. When seeking maximum capacity, a benchmark typically adds network and I/O capacity until CPU is exhausted.

A typical server does not assign priorities to incoming requests based on type of request or originating client. It processes incoming requests within the context of several *nfsd* daemons. The number of *nfsds* controls the number of NFS requests that a server can work on concurrently.

4.3. NFS Server Writes and Stable Storage

An NFS server should commit any modified data to stable storage before responding to the client that the request is complete [SAND85]. If a server is not following this rule, then it is not living up to its part of the agreement implicit in the NFS crash recovery design. An asynchronous operation carries with it the promise to fully complete that operation at some later time. The NFS protocol contains no provisions for recalling past promises (which is precisely why crash recovery is so easy). Without a way to recall past unkept promises, a server should not make them. Traditionally, this meant that the server had to fully commit all data and modified metadata associated with the write operation to disk before responding.

The cost, in latency and server loading, of meeting this requirement is significant, and has led to varying vendor reactions and developments:

- The appearance of filesystem accelerators consisting of s/w and non-volatile RAM (NVRAM) h/w, e.g. Prestoserve [MORA90], [PRES93], that address the issues of latency and, to some degree, I/O loading (but not server CPU loading) while operating within the stable storage paradigm.
- Some vendors have chosen to make async NFS writes, aka "dangerous mode", an administrative option. In this scenario, the server responds after data is committed to volatile storage, whether main memory or disk controller, an option. Some vendors have made this mode the default behavior and supply an uninterruptible power supply (UPS) with their servers. Some vendors simply make it the default behavior without UPS.

After much discussion, SPEC has arrived at the following requirement for reporting SPEC/LADDIS *baseline* results [SPEC93]. A baseline conforming NFS server must:

- commit all write data and associated metadata through to disk,
- or to other stable storage (e.g. NVRAM) that is recovered and flushed to disk after server failure,
- or to volatile RAM (main memory) if powered by UPS *and* if the OS supports data recovery following power and h/w failures.

This paper assumes that the server conforms to the SPEC baseline reporting requirements with respect to stable storage guarantees by using one of the first two operational techniques listed above.

4.4. Local Filesystem Operations

This paper assumes that the filesystem being served is of BSD FFS vintage [MCKU84], as is the case with the reference port. My work was done using a BSD 4.3 filesystem (UFS) with extensions that cluster reads and writes into larger device request sizes (up to 64K) in a manner similar to that described in [MCVO91].

For each remote write request, at least one, and possibly two or three synchronous disk operations must be performed by the server before a response can be sent to the client indicating that the request has been completed. At the very least, the data block in question must be written. If the write increased the size of the file, or on-disk structures have changed (e.g. adding a direct block to fill a "hole" in the file), then the block containing the inode must be written. Finally, if an indirect block was modified, then it too must be written before responding.

The reference port makes a special case for the file modify time in the inode. If modify time is the only item changed in the inode as a result of a write operation, i.e. a write to a previously allocated block, then the inode update to disk is performed asynchronously. This (the file modification time) is one promise that the server may not keep; this risk is taken for the benefits of better performance.

5. A Case Study

Let's assume we are using a network monitoring tool (like tcpdump(8)) to observe the network while client process C, described earlier, writes a reasonably large file, and for the sake of simplicity assume that the client is doing nothing else, and that this is a private network. Finally, let's assume that the client and a typical server, S, are well matched enough so that we don't have any lost requests or responses, timeouts, etc.

As C begins to process, network traffic will resemble a freight train of 8K (actually a little larger due to protocol headers, etc.) datagrams fragmented into transport units directed toward S. The total length of the train is dependent upon the number of biods available for use by C. Many biods mean a very long train. The time needed for S to process the first write request (which was handed off to a biod) is typically larger than the time needed for C to generate more requests. Several, and perhaps many, writes can build up on the server side before the first response is sent. Process C blocks when the biods all become busy. The traffic direction now changes to one of responses directed toward C at intervals of time needed by S to process the buffered write requests. No further requests are generated by C until a response is received for the last write request; the traffic direction now switches again as C resumes processing. A cycle of these uni-directional traffic shifts continues as described above until the entire file has been transferred.

The left half of Figure 1. (the communication between Client and Standard Server) depicts a portion of this traffic flow for the 4 biod case, after the client has written about 100K of data.

If the file in question was newly created and of size $N \times 8K$, and that the server filesystem was of blocksize 8K, then the total number of server disk operations was roughly $3N$:

- N data writes,
- N inode block updates,
- N indirect block updates (minus n direct blocks)

6. Write Gathering

The write gathering technique described here attempts to reduce the $3N$ disk operations described in the previous section (*A Case Study*) to as close to N as it can. With an underlying filesystem that supports clustering [MCVO91], and a sequential write pattern, the number of disk operations can be reduced far below N . Optimal

DEC 3500 client, 3800 server, rz26 disk, FDDI network

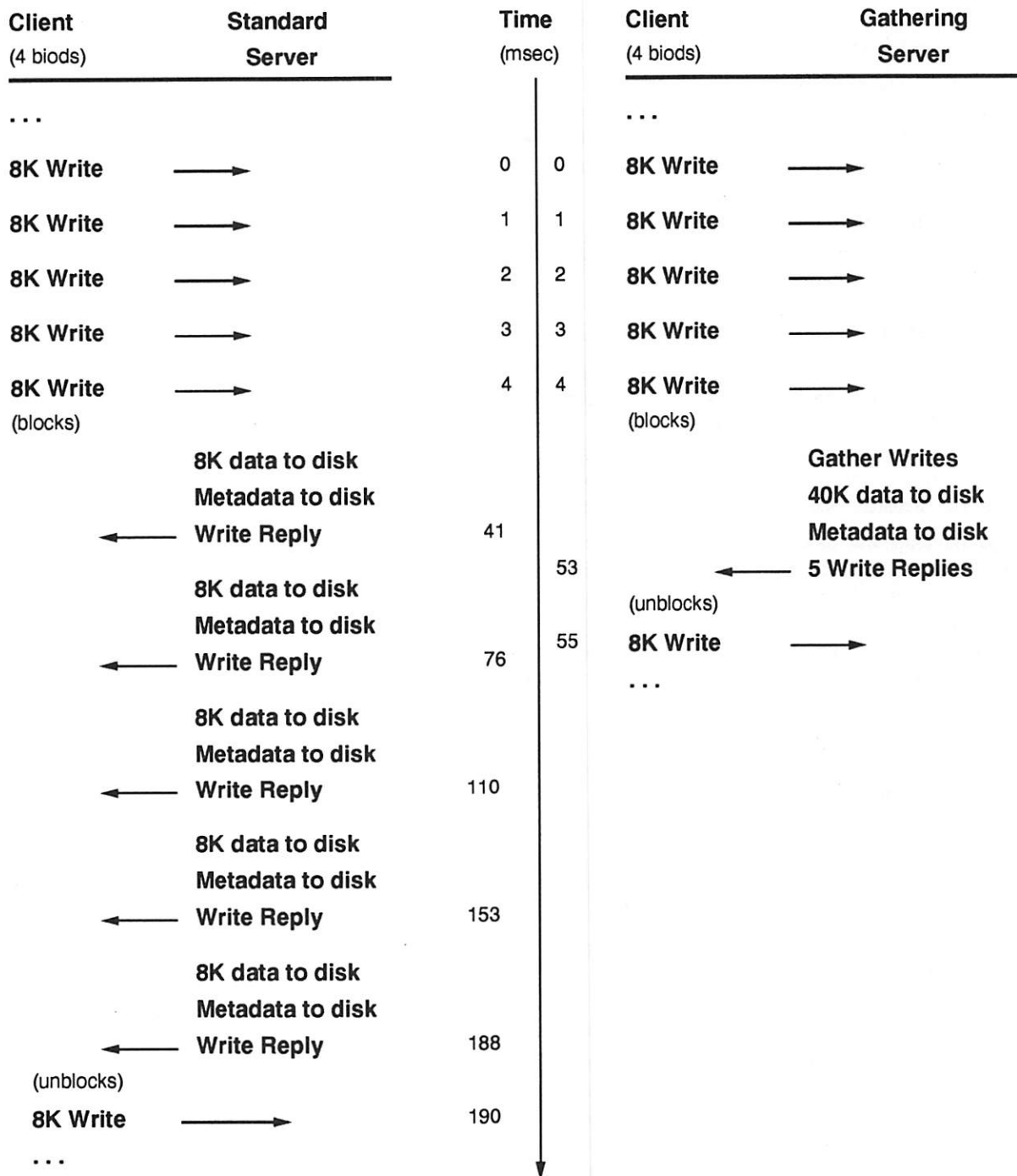


Figure 1. Write Gathering NFS Server Comparison
Sequential File Writer, >100K Into File

write gathering occurs when the minimal number of disk transactions is generated for a particular load with a particular underlying filesystem. The description of write gathering that follows assumes an underlying server filesystem (e.g. UFS) that supports write clustering.

The object of this technique, from the perspective of an `nfsd`, is to avoid doing the metadata update. An `nfsd`, `D`, tries to assign this task to some other `nfsd` which will send `D`'s response as well as its own.

If we use the network monitoring example from the previous section with a write gathering server instead, the change we see is in the timing of the replies from server to client. The server "digests" all the write requests and sends replies in first in, first out order; all the replies have the same file modify time in the returned file attributes. The total elapsed time from first request to last reply is less due to disk efficiencies (fewer, larger writes, fewer seeks).

The right half of Figure 1. (the communication between Client and Gathering Server) depicts a portion of this traffic flow for the 4 biod case, after the client has written about 100K of data. In this case, optimal write gathering of 3 disk transactions has been achieved.

6.1. NFS/RPC Architectural Changes

In a reference port server, each `nfsd` process makes a call into kernel level RPC (the `svc_run()` routine) with a transport handle (used to store client and request information) and the address of an NFS layer routine (`rfs_dispatch()`) that dispatches incoming requests to appropriate server layer action routines (e.g. `rfs_write()` for incoming write requests). `Svc_run()` does not return until the `nfsd` process dies. Information needed to send a response is stored in the transport handle which is tied to the `nfsd` process that started work on the request. When the action routine returns, `rfs_dispatch()` sends the reply to the client and returns to `svc_run()`.

This architecture was modified so that one `nfsd` can process a write request to a certain point and then arrange for another `nfsd` to send the reply. The first `nfsd` is then free to look for other work. It returns an indication to `rfs_dispatch()` that the reply is delayed, who conveys this back into the RPC layer; another transport handle is taken from a cache of free handles and the `nfsd` process is ready to process other work, possibly another write to the same file. This architecture allows optimal write gathering to take place with as few as one `nfsd` available on the server; this is an architecture that should scale well for large servers with many active client writers.

6.2. New Data Structures

A global array of `nfsd` state was created so that one `nfsd` can ascertain the state of others. Most notably, whether another `nfsd` is processing a write, and to which file, and to which offset and length, and at what stage the `nfsd` is in the processing of a write. With this information an `nfsd` can decide whether it can leave the task of metadata update to another, "following" `nfsd`.

As mentioned above, a cache of free transport handles was implemented, along with data structures that package up active write requests for handoff and a queue of these active requests.

OSF/1 provides a vnode spin lock, but not a sleep lock. I added a vnode sleep lock for `nfsd` serialization and synchronization.

6.3. Prestoserve/NVRAM Acceleration vs. Disks

NVRAM accelerated disks have radically different latency properties from non-accelerated ones. With a non-accelerated disk, the best policy is to cache/coalesce/cluster writes within UFS as long as possible in the hope of doing fewer, larger disk writes. With a Prestoserve accelerated disk, the best policy is to get individual writes down to the Prestoserve driver (Presto) as soon as possible. Presto does its own clustering. With Presto there is no benefit to holding writes within UFS as when using a non-accelerated disk; in fact it is less efficient because Presto can drive disks asynchronously and in parallel with NFS write and reply processing. Also, due to the relatively small size of the NVRAM cache (typically one or more MB), Presto may decline to accept requests above a certain size (typically 8K), resulting in performance that degrades to underlying disk speed.

The observations above lead to a duality within the server write layer; it was modified to query Presto as to acceleration state of a filesystem (on/off) and operate in different ways depending upon state.

6.4. Filesystem Hints Through VFS

The VFS (GFS for ULTRIX) layer was modified so that the server layer could send hints to the underlying filesystem.

If the filesystem is accelerated, the VOP_WRITE routine is called with the IO_SYNC and the (new) IO_DATAONLY flags, delivering the data to Presto but delaying any metadata copies (and consumption of CPU cycles). If the filesystem is not accelerated, the VOP_WRITE routine is called with the (new) IO_DELAYDATA flag, freeing UFS to choose its own clustering policy (and perhaps starting an asynchronous write). Metadata is flushed via a call to VOP_FSYNC with the FWRITE and the (new) FWRITE_METADATA flags to ensure that only the inode and indirect blocks are flushed.

For non-accelerated disks, when write gathering terminates (described below) and metadata is flushed, data blocks are flushed via a call to the (new) VOP_SYNCDATA routine with beginning and ending offsets as hints.

6.5. The Socket Buffer

With Prestoserve acceleration, there is often no I/O event associated with a VOP_WRITE, and the nfsd process D does not block. If it does not block there is no opportunity for write requests to be delivered to other nfsds even if they have been placed on the socket buffer. A routine (the mbuf hunter) was written (hacked) to scan the socket buffer searching for NFS writes for a given file and returning true/false. The mbuf hunter is a gross violation of kernel layering, but with a fast server this technique is often a win (and thus the hack has redeeming virtue).

6.6. Procrastinate

(pro-kras'-ti-nate) *verb.* To put off, esp. habitually, doing something until a future time. --pro-cras'ti-na'tor.

This is probably the most controversial aspect of write gathering. The technique injects a small amount of latency into the processing cycle, hoping to give another write request an opportunity to arrive at the server. This latency is approx. (modulo h/w clock accuracy) 8 msec for Ethernet or multi-segment requests and 5 msec for FDDI based requests. These values were derived via empirical lab experiments. Private networks were used to ascertain values that allowed for optimal gathering, and a tick or two was added for conservatism. Also, systems in more general use have been monitored for gathering success rates (but conclusions are sometimes difficult to draw without detailed knowledge of request patterns).

I wish I could say I know how to calculate the "right" number, but I don't. Clearly there is room for more work here. When write gathering "fails" because the server didn't wait long enough, it falls back to typical, standard, server processing behavior.

The implementation described in [SIVA93] takes the first write encountered and sends it to disk, using this operation as "the latency device" which gives more write requests time to arrive at the server. I considered this approach early on and abandoned it for two reasons:

First, running spindles with a request pattern of anything other than a pure stream of large requests is sub-optimal in both drive throughput and CPU utilization. It's difficult to approach raw device speeds with 8K requests in the device queue without dangerous mode operation (controller write caching). And ignoring protocol requirements, with this there are still too many trips through the driver.

Second, it just won't work with NVRAM acceleration where the first write is done faster than other writes can arrive.

6.7. Order of Replies

At first it seemed a good idea to order replies in LIFO order so as to wake up a blocked client process C (see *A Case Study*, above). In fact, this yielded dismal results for the common file transfer case because C would generate fewer (and maybe none for a fast client) writes before blocking the next time if there were not enough boids available. LIFO was abandoned for FIFO; this optimized the case of a single sequential file writer. It also seems reasonable to free up boids on the client for other work (by other processes) sooner, in the multiprocessing client case. Note that FIFO is the order used by typical (standard) existing servers, and this is not a change in behavior.

6.8. The Write Gathering Algorithm

D is an `nfds` handed a write request:

Hand off data to UFS via `VOP_WRITE` (as described above).

Do

Look for another `nfds` blocked on the same `vnode`.

If one is,

Add write descriptor to the active write queue.

Return to `rfs_dispatch()` with a reply-pending code.

Else search the socket buffer for another write request to the same file.

If there is,

Add write descriptor to the active write queue.

Return to `rfs_dispatch()` with a reply-pending code.

Sleep (procrastinate) for a transport dependent interval.

While not procrastinating more than once.

Become the metadata writer and assume responsibility for this file:

Flush this and other data for active writes via `VOP_SYNCDATA`.

Flush the metadata via `VOP_FSYNC`.

Send all pending replies for the file to the client.

Return to `rfs_dispatch()` with a reply-done code.

6.9. Duplicate Requests, Stale File Handles, Etc.

Stale file handles are client references to files that no longer exist. See [JUSZ89] for a discussion of duplicate NFS requests.

The existence of these requests, in the socket buffer e.g., could have caused `nfds`s to delay their replies. The implementor must not be too hasty discarding duplicates, etc. (meaning I was at first!); this could result in orphaned writes on the active write queue with no meta data writer to send replies.

6.10. What About Dumb PCs?

Single threaded PCs (or clients with no biods, or clients that emit a single write every once in a while) are the worst case for write gathering. There is added processing and latency for no gain. The actual measured loss from the client's perspective (easily simulated by killing all biods) is about 15% in throughput (over Ethernet) with a reasonably quick server and a quick single threaded client (see *Results*). This loss decreases in significance as slower clients are used.

6.11. What About Random Access?

The write gathering algorithm does not assume an ordering on the delivery of writes. A grouping of random access writes will accrue the same benefits of metadata amortization as a grouping of sequential access writes. The clustering of data blocks, and the resultant number of disk transactions for them, is an underlying filesystem issue.

7. Results

7.1. NFS Sequential Write Bandwidth

This section contains the results of experiments where a 10MB file is written over private Ethernet and FDDI networks with and without write gathering in effect and while varying the number of client biods. The server used 8 `nfds`s. For Ethernet, the client and server are DEC 3400s, the server is using an rz26 (1GB SCSI) disk. For FDDI, I used somewhat faster systems (for no better reason than that is the way my lab is set up), the client is a DEC 3500, the server is a DEC 3800, using either one, or a stripe set of three rz26 disk(s), as labeled.

Table 1. shows the experiment, without write gathering, being limited by spindle speeds for 8K transfers. Table 1. also shows the cost of write gathering in the worst case (no biods) as 15%, and the gain in the best case (15 biods) as 228%. For the 7 biods case the gain is 145%.

Table 1. NFS 10MB file copy: Ethernet

# of Client BIODs	0	3	7	11	15
Without Write Gathering					
client write speed (KB/sec.)	165	194	201	203	205
server cpu util. (%)	9	11	11	12	12
server disk (KB/sec)	480	570	590	590	590
server disk (trans/sec)	61	71	72	73	74
With Write Gathering					
client write speed (KB/sec.)	140	375	493	575	674
server cpu util. (%)	7	14	16	19	21
server disk (KB/sec)	415	550	610	660	750
server disk (trans/sec)	52	47	24	31	21

What is not obvious in Table 1. is that write gathering is conserving server CPU (by saved UFS and driver trips); this is hidden by the greater throughput. Table 2. shows the same disk under Prestoserve acceleration, where the latencies involved are NVRAM copies instead of moving head disk operations. With Prestoserve (where Presto is clustering and handling the underlying disk efficiently), write gathering increases CPU efficiency at the expense of some client throughput. The 7 biod case shows a decrease of 26% in server utilization at the cost of 15% in client throughput. The bulk of this savings is the reduction of metadata operations through UFS and Presto.

Table 2. NFS 10MB file copy: Ethernet, Presto

# of Client BIODs	0	3	7	11	15
Without Write Gathering					
client write speed (KB/sec.)	809	1025	1080	1103	1112
server cpu util. (%)	30	38	41	42	43
server disk (KB/sec)	789	1004	1080	1104	1080
server disk (trans/sec)	7	8	9	9	9
With Write Gathering					
client write speed (KB/sec.)	439	787	915	959	991
server cpu util. (%)	18	26	30	32	34
server disk (KB/sec)	430	770	885	949	985
server disk (trans/sec)	4	7	7	9	8

Now we change the configuration by moving to an FDDI network, reducing network latencies, server CPU overhead due to packet reassembly, etc. In Table 3., for the 15 biod case, we see the server processing NFS writes at about 1M/sec. without Presto acceleration.

Table 3. NFS 10MB file copy: FDDI

# of Client BIODs	0	3	7	11	15
Without Write Gathering					
client write speed (KB/sec.)	207	209	207	209	208
server cpu util. (%)	6	6	6	6	6
server disk (KB/sec)	605	610	605	615	615
server disk (trans/sec)	76	77	76	75	77
With Write Gathering					
client write speed (KB/sec.)	177	534	846	876	1085
server cpu util. (%)	6	9	10	11	12
server disk (KB/sec)	520	780	975	1000	1175
server disk (trans/sec)	66	65	38	45	33

Table 4. shows the rz26 disk being driven at the raw device write bandwidth limit for 64K transfers under Presto acceleration.

Table 4. NFS 10MB file copy: FDDI, Presto

# of Client BIODs	0	3	7	11	15
Without Write Gathering					
client write speed (KB/sec.)	1883	1898	1863	1900	1918
server cpu util. (%)	33	34	35	35	34
server disk (KB/sec)	1833	1848	1844	1844	1900
server disk (trans/sec)	16	16	15	15	16
With Write Gathering					
client write speed (KB/sec.)	927	1850	1888	1895	1894
server cpu util. (%)	13	24	28	27	27
server disk (KB/sec)	910	1745	1889	1882	1867
server disk (trans/sec)	8	17	16	16	16

This experiment shown by Table 4. was limited by spindle speeds again similar to Table 1., but the client is operating at near maximal device bandwidth. Tables 5. and 6., where a 3 drive stripe set is used, shows the effect upon the configuration of increasing disk bandwidth. In Table 5., for the 15 biod case, write speed has increased with write gathering by 262%; CPU utilization has increased 36%. In Table 6., for the 15 biod case, write speed has decreased with write gathering by 20%, but CPU utilization has also decreased by 40% (relative to Table 5.).

Table 5. NFS 10MB file copy: FDDI, 3 striped drives

# of Client BIODs	0	3	7	11	15	19	23
Without Write Gathering							
client write speed (KB/sec.)	200	275	299	304	308	308	313
server cpu util. (%)	7	10	11	11	11	11	12
server disks (KB/sec)	560	827	865	895	879	921	927
server disks (trans/sec)	72	104	110	112	111	115	117
With Write Gathering							
client write speed (KB/sec.)	187	574	814	987	1115	1287	1618
server cpu util. (%)	7	11	13	15	15	18	22
server disks (KB/sec)	560	785	984	1109	1225	1384	1695
server disks (trans/sec)	71	72	60	65	67	71	74

Table 6. NFS 10MB file copy: FDDI, Presto, 3 striped drives

# of Client BIODs	0	3	7	11	15	19	23
Without Write Gathering							
client write speed (KB/sec.)	2102	3403	3394	3503	3474	3360	3342
server cpu util. (%)	40	66	69	68	70	71	70
server disks (KB/sec)	2067	3146	3515	3349	3305	3575	3445
server disks (trans/sec)	47	71	80	77	76	80	78
With Write Gathering							
client write speed (KB/sec.)	1015	2144	2649	2775	2754	3078	3048
server cpu util. (%)	6	29	42	42	42	43	46
server disks (KB/sec)	1008	2143	2644	2724	2685	2501	2627
server disks (trans/sec)	22	49	61	62	63	59	63

7.2. SPEC/LADDIS Results

Writes are a small (15%) portion of Nfsstone [LEGA89] and SPEC/LADDIS [WITT93] workloads, but they are expensive to process. This technique yielded a positive effect on SPEC/LADDIS server throughput and latency via its server efficiency gains. Figure 2. shows an increase of 13% in server capacity along with an 11% reduction

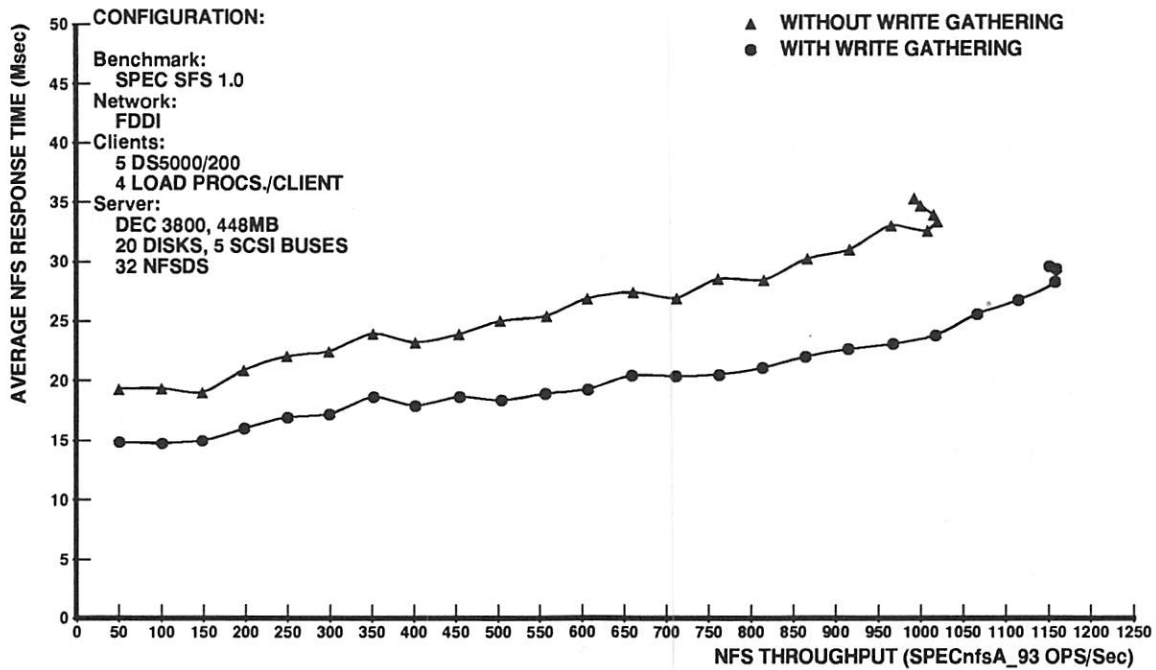


FIGURE 2. DEC 3800 SPEC SFS 1.0 BASELINE RESULTS

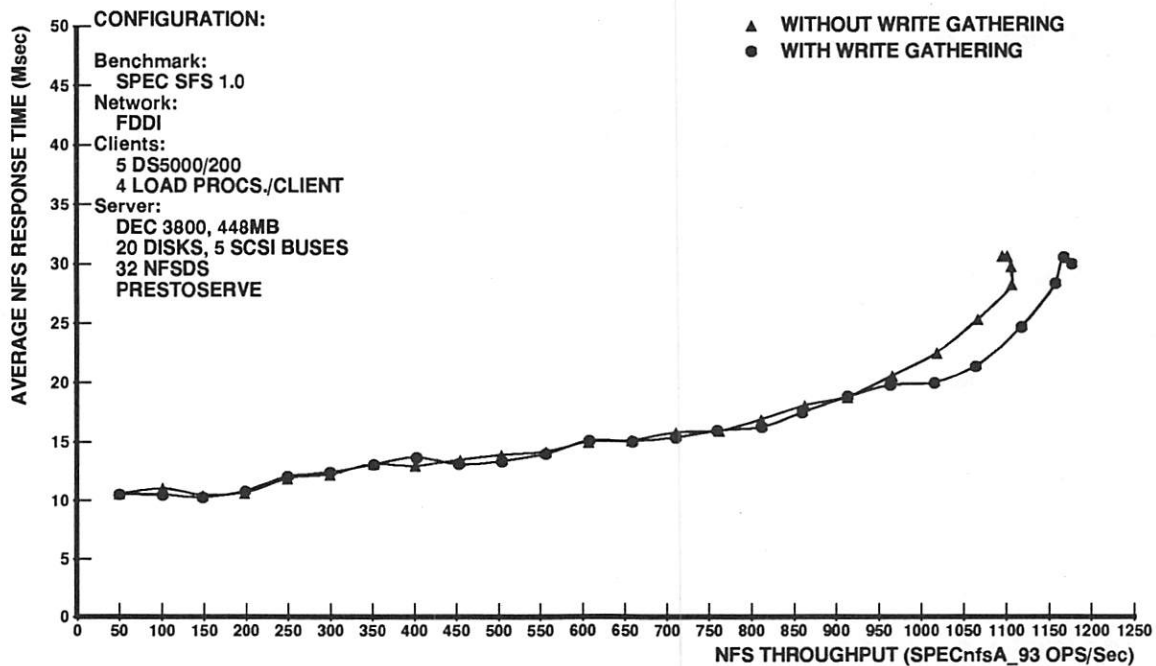


FIGURE 3. DEC 3800 PRESTOSERVE SPEC SFS 1.0 BASELINE RESULTS

in average latency for a DEC 3800 server using write gathering as measured by SPEC SFS 1.0 (LADDIS). Figure 3. shows more modest, but still positive, gains for the same configuration with Prestoserve in effect.

8. Future Work

The NFS Version 3 protocol supports reliable asynchronous writes in addition to the Version 2 stable storage write semantics. Many V3 clients may opt for the simpler kernel implementation of V2 write semantics. This will ensure the usefulness of write gathering in a V3 environment. It will be interesting to see if this technique applies itself in some new way in a mixed environment of V2 clients, V3 clients using V2 semantics, and V3 clients using reliable asynchronous writes.

The worst case scenario for the current write gathering algorithm is with single threaded clients, such as dumb PCs, where there is never an opportunity to gather writes, and the CPU effort and added latency is a loss. This is a tradeoff that should be considered by the implementor and/or server administrator (it's easy to turn write gathering off). Some might say that it's doubtful whether a truly dumb (spelled slow) PC can tell the difference with an otherwise fast server, but it would be very nice to say that there is no performance penalty for single threaded writers. Jeff Mogul has suggested a scheme where the server builds a small database of "learned" information about individual clients, and uses this to direct gathering behavior. Clearly there is room for improvement here.

9. Conclusions

Write gathering can help to improve server capacity. It takes a lot of CPU cycles to run the disk driver and field device interrupts and/or copy data to NVRAM. If the NFS server layer can avoid some disk writes it is a big win; big enough to make it worth the gamble of spending some CPU cycles trying to be clever and avoid the writes.

Write gathering improves write bandwidths. Write gathering plays well with UFS clustering; it is possible to get closer to raw device speeds with NFS writes because fewer, larger, disk writes are done and fewer seeks and missed rotations are experienced. I was able to achieve a significant increase in client write speeds while at the same time often reducing server loading.

Write gathering exploits client behavior (i.e. biods) that has been typical in workstation clients since NFS was introduced. Write gathering efficiencies increase as the number of biods increase and has led some vendors to increase their defaults. With 8K transfers and UFS 64K clustering, 7 biods result in the ideal case on the server of one 64K disk write for data and one metadata update per set of (application + 7 biods) requests. The addition of more biods on the client may increase throughput if the carrying capacity of the network/server can support it (the server socket buffer, e.g., is a limit: DEC OSF/1 currently uses a maximum of .25M for socket buffering). As a rule of thumb, I don't recommend more than 7 biods for general purpose/heavily used networks.

The work described here was done for a filesystem with BSD FFS vintage on-disk structure [MCKU84]. Log-based and log-structured local filesystems are becoming more popular. Although on-disk structures may vary, the NFS stable storage requirement imposes a relationship between disk performance and server write performance. Hopefully this description of gathering network originated write requests in a network service layer, and the passing of hints from this layer to the underlying local filesystem, will prove useful with other filesystem types.

The implementation described here was made part of the ULTRIX Version 4.3 and DEC OSF/1 Version 1.2 operating systems.

10. Acknowledgements

Thanks go to: Drew Perkins for the NFS Connectathon demo that got me going. It was similar in impact to an earlier Connectathon demo of Prestoserve. [MCVO91] for providing the UFS clustering inspiration. Paul Shaughnessy for providing/adapting our UFS implementation. Brian Nadeau and Allen Rollow for providing the disk striping driver used in *Results*. Jeff Mogul for the nudge to publish and general support. Charlie Briggs, as usual, was a sounding board and provider of clear thinking during the project; he also provided the mbuf hunter.

11. References

[BROW90] Ted Smalley Brown, "Software update speeds NFS write process on server", trade publication article, Epoch Systems Inc.'s HyperWrite product, *Digital Review*, v7, n30 (August 6, 1990), p. 17.

- [JUSZ89] Chet Juszczak, "Improving the Performance and Correctness of an NFS Server", *Proceedings Winter Usenix 1989*, San Diego, CA, 53-63, January 1989.
- [LEGA89] Sandberg, R., "nhfsstone" NFS load generating program, Legato Systems, Inc., Palo Alto, CA.
- [MCKU84] McKusick, M. K., et. al., "A Fast File System for UNIX", *ACM Transactions on Computer Systems*, 2(3), August 1984, 181-197.
- [MCVO91] McVoy, L., Kleiman, S., "Extent-like Performance from a Unix File System", *Proceedings Winter Usenix 1991*, Dallas, TX, 33-43, January, 1991.
- [MORA90] Moran, J., Sandberg, R., Coleman, D., Kepecs, J., Lyon, B., "Breaking Through the NFS Performance Barrier", *Proceedings of the 1990 Spring European Unix Users Group*, Munich, Germany, 199-206, April 1990
- [PRES93] Digital Equipment Corporation, Maynard, MA, "Guide to Prestoserve", *DEC OSF/1 Prestoserve Product Documentation*, Order number AA-PQT0A-TE, March 1993.
- [SAND85] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., Lyon, B., "Design and Implementation of the Sun Network Filesystem", *Summer 1985 Usenix Conference Proceedings*, Portland, OR, 119-130, June 1985.
- [SIVA93] Sivaprakasam, Suresh, "Performance Enhancements in SunOS NFS", *Technical Report TR 93-18*, State University of New York, Buffalo Computer Science Dept., May, 1993.
- [SPEC93] Standard Performance Evaluation Corporation, "SPEC Run Rules for SFS Release 1.0", *SPEC Steering Committee Memorandum*, June 15, 1993, Sec. 7.1.1.4, p. 6, Contained in the SFS 1.0 Release Materials.
- [WITT93] Wittle, M, Keith, B., "LADDIS: The Next Generation in NFS File Server Benchmarking", *Summer 1993 Usenix Conference Proceedings*, Cincinnati, OH, 111-128, June, 1993.

12. Author Information

Chet Juszczak is a Consultant Engineer in the Unix Software Group at DEC where he has been working on NFS and file server performance since 1985. He was involved with the definition of the NFS Version 3 protocol. He got his M.S. in C.S. at the University of Michigan in 1983. Reach him electronically at chet@zk3.dec.com or via U.S. Mail at Digital Equipment Corp., 110 Spit Brook Rd., Nashua NH 03062.

13. Trademarks

DEC and ULTRIX are trademarks of Digital Equipment Corporation.

Ethernet is a trademark of Xerox Corporation.

OSF/1 is a trademark of Open Software Foundation, Inc.

NFS and Solaris are trademarks of Sun Microsystems, Inc.

Prestoserve is a trademark of Legato Systems, Inc.

SPEC is a trademark of the Standard Performance Evaluation Corporation.

UNIX is a registered trademark of Unix Systems Laboratories, Inc.

Not Quite NFS, Soft Cache Consistency for NFS

Rick Macklem
University of Guelph

Abstract

There are some constraints inherent in the NFS protocol that result in performance limitations for high performance workstation environments. This paper discusses an NFS-like protocol named Not Quite NFS (NQNFS), designed to address some of these limitations. This protocol provides full cache consistency during normal operation, while permitting more effective client-side caching in an effort to improve performance. There are also a variety of minor protocol changes, in order to resolve various NFS issues. The emphasis is on observed performance of a preliminary implementation of the protocol, in order to show how well this design works and to suggest possible areas for further improvement.

1. Introduction

It has been observed that overall workstation performance has not been scaling with processor speed and that file system I/O is a limiting factor [Ousterhout90]. Ousterhout notes that a principal challenge for operating system developers is the decoupling of system calls from their underlying I/O operations, in order to improve average system call response times. For distributed file systems, every synchronous Remote Procedure Call (RPC) takes a minimum of a few milliseconds and, as such, is analogous to an underlying I/O operation. This suggests that client caching with a very good hit ratio for read type operations, along with asynchronous writing, is required in order to avoid delays waiting for RPC replies. However, the NFS protocol requires that the server be stateless¹ and does not provide any explicit mechanism for client cache consistency, putting constraints on how the client may cache data. This paper describes an NFS-like protocol that includes a cache consistency component designed to enhance client caching performance. It does provide full consistency under normal operation, but without requiring that hard state information be maintained on the server. Design tradeoffs were made towards simplicity and high performance over cache consistency under abnormal conditions. The protocol design uses a variation of Leases [Gray89] to provide state on the server that does not need to be recovered after a crash.

The protocol also includes changes designed to address other limitations of NFS in a modern workstation environment. The use of TCP transport is optionally available to avoid the pitfalls of Sun RPC over UDP transport when running across an internetwork [Nowicki89]. Kerberos [Steiner88] support is available to do proper user authentication, in order to provide improved security and arbitrary client to server user ID mappings. There are also a variety of other changes to accommodate large file systems, such as 64bit file sizes and offsets, as well as lifting the 8Kbyte I/O size limit. The remainder of this paper gives an overview of the protocol, highlighting performance related components, followed by an evaluation of resultant performance for the 4.4BSD implementation.

2. Distributed File Systems and Caching

Clients using distributed file systems cache recently-used data in order to reduce the number of synchronous server operations, and therefore improve average response times for system calls. Unfortunately, maintaining consistency between these caches is a problem whenever write sharing occurs; that is, when a process on a client writes to a file and one or more processes on other client(s) read the file. If the writer closes the file before any reader(s) open the file for reading, this is called sequential write sharing. Both the Andrew ITC file system

¹ The server must not require any state that may be lost due to a crash, to function correctly.

[Howard88] and NFS [Sandberg85] maintain consistency for sequential write sharing by requiring the writer to push all the writes through to the server on close and having readers check to see if the file has been modified upon open. If the file has been modified, the client throws away all cached data for that file, as it is now stale. NFS implementations typically detect file modification by checking a cached copy of the file's modification time; since this cached value is often several seconds out of date and only has a resolution of one second, an NFS client often uses stale cached data for some time after the file has been updated on the server.

A more difficult case is concurrent write sharing, where write operations are intermixed with read operations. Consistency for this case, often referred to as "full cache consistency," requires that a reader always receives the most recently written data. Neither NFS nor the Andrew ITC file system maintain consistency for this case. The simplest mechanism for maintaining full cache consistency is the one used by Sprite [Nelson88], which disables all client caching of the file whenever concurrent write sharing might occur. There are other mechanisms described in the literature [Kent87a, Burrows88], but they appeared to be too elaborate for incorporation into NQNFS (for example, Kent's requires specialized hardware). NQNFS differs from Sprite in the way it detects write sharing. The Sprite server maintains a list of files currently open by the various clients and detects write sharing when a file open request for writing is received and the file is already open for reading (or vice versa). This list of open files is hard state information that must be recovered after a server crash, which is a significant problem in its own right [Mogul93, Welch90].

The approach used by NQNFS is a variant of the Leases mechanism [Gray89]. In this model, the server issues to a client a promise, referred to as a "lease," that the client may cache a specific object without fear of conflict. A lease has a limited duration and must be renewed by the client if it wishes to continue to cache the object. In NQNFS, clients hold short-term (up to one minute) leases on files for reading or writing. The leases are analogous to entries in the open file list, except that they expire after the lease term unless renewed by the client. As such, one minute after issuing the last lease there are no current leases and therefore no lease records to be recovered after a crash, hence the term "soft server state."

A related design consideration is the way client writing is done. Synchronous writing requires that all writes be pushed through to the server during the write system call. This is the simplest variant, from a consistency point of view, since the server always has the most recently written data. It also permits any write errors, such as "file system out of space" to be propagated back to the client's process via the write system call return. Unfortunately this approach limits the client write rate, based on server write performance and client/server RPC round trip time (RTT).

An alternative to this is delayed writing, where the write system call returns as soon as the data is cached on the client and the data is written to the server sometime later. This permits client writing to occur at the rate of local storage access up to the size of the local cache. Also, for cases where file truncation/deletion occurs shortly after writing, the write to the server may be avoided since the data has already been deleted, reducing server write load. There are some obvious drawbacks to this approach. For any Sprite-like system to maintain full consistency, the server must "callback" to the client to cause the delayed writes to be written back to the server when write sharing is about to occur. There are also problems with the propagation of errors back to the client process that issued the write system call. The reason for this is that the system call has already returned without reporting an error and the process may also have already terminated. As well, there is a risk of the loss of recently written data if the client crashes before the data is written back to the server.

A compromise between these two alternatives is asynchronous writing, where the write to the server is initiated during the write system call but the write system call returns before the write completes. This approach minimizes the risk of data loss due to a client crash, but negates the possibility of reducing server write load by throwing writes away when a file is truncated or deleted.

NFS implementations usually do a mix of asynchronous and delayed writing but push all writes to the server upon close, in order to maintain open/close consistency. Pushing the delayed writes on close negates much of the performance advantage of delayed writing, since the delays that were avoided in the write system calls are observed in the close system call. Akin to Sprite, the NQNFS protocol does delayed writing in an effort to achieve good client performance and uses a callback mechanism to maintain full cache consistency.

3. Related Work

There has been a great deal of effort put into improving the performance and consistency of the NFS protocol. This work can be put in two categories. The first category are implementation enhancements for the NFS protocol and the second involve modifications to the protocol.

The work done on implementation enhancements have attacked two problem areas, NFS server write performance and RPC transport problems. Server write performance is a major problem for NFS, in part due to the requirement to push all writes to the server upon close and in part due to the fact that, for writes, all data and meta-data must be committed to non-volatile storage before the server replies to the write RPC. The Prestoserve™[†] [Moran90] system uses non-volatile RAM as a buffer for recently written data on the server, so that the write RPC replies can be returned to the client before the data is written to the disk surface. Write gathering [Juszczak94] is a software technique used on the server where a write RPC request is delayed for a short time in the hope that another contiguous write request will arrive, so that they can be merged into one write operation. Since the replies to all of the merged writes are not returned to the client until the write operation is completed, this delay does not violate the protocol. When write operations are merged, the number of disk writes can be reduced, improving server write performance. Although either of the above reduces write RPC response time for the server, it cannot be reduced to zero, and so, any client side caching mechanism that reduces write RPC load or client dependence on server RPC response time should still improve overall performance. Good client side caching should be complementary to these server techniques, although client performance improvements as a result of caching may be less dramatic when these techniques are used.

In NFS, each Sun RPC request is packaged in a UDP datagram for transmission to the server. A timer is started, and if a timeout occurs before the corresponding RPC reply is received, the RPC request is retransmitted. There are two problems with this model. First, when a retransmit timeout occurs, the RPC may be redone, instead of simply retransmitting the RPC request message to the server. A recent-request cache can be used on the server to minimize the negative impact of redoing RPCs [Juszczak89]. The second problem is that a large UDP datagram, such as a read request or write reply, must be fragmented by IP and if any one IP fragment is lost in transit, the entire UDP datagram is lost [Kent87]. Since entire requests and replies are packaged in a single UDP datagram, this puts an upper bound on the read/write data size (8 kbytes).

Adjusting the retransmit timeout (RTT) interval dynamically and applying a congestion window on outstanding requests has been shown to be of some help [Nowicki89] with the retransmission problem. An alternative to this is to use TCP transport to delivery the RPC messages reliably [Macklem90] and one of the performance results in this paper shows the effects of this further.

Srinivasan and Mogul [Srinivasan89] enhanced the NFS protocol to use the Sprite cache consistency algorithm in an effort to improve performance and to provide full client cache consistency. This experimental implementation demonstrated significantly better performance than NFS, but suffered from a lack of crash recovery support. The NQNFS protocol design borrowed heavily from this work, but differed from the Sprite algorithm by using Leases instead of file open state to detect write sharing. The decision to use Leases was made primarily to avoid the crash recovery problem. More recent work by the Sprite group [Baker91] and Mogul [Mogul93] have addressed the crash recovery problem, making this design tradeoff more questionable now.

Sun has recently updated the NFS protocol to Version 3 [SUN93], using some changes similar to NQNFS to address various issues. The Version 3 protocol uses 64bit file sizes and offsets, provides a Readdir_and_Lookup RPC and an access RPC. It also provides cache hints, to permit a client to be able to determine whether a file modification is the result of that client's write or some other client's write. It would be possible to add either Spritely NFS or NQNFS support for cache consistency to the NFS Version 3 protocol.

4. NQNFS Consistency Protocol and Recovery

The NQNFS cache consistency protocol uses a somewhat Sprite-like [Nelson88] mechanism, but is based on Leases [Gray89] instead of hard server state information about open files. The basic principle is that the server disables client caching of files whenever concurrent write sharing could occur, by performing a server-to-client callback, forcing the client to flush its caches and to do all subsequent I/O on the file with synchronous RPCs. A Sprite server maintains a record of the open state of files for all clients and uses this to determine when concurrent write sharing might occur. This *open state* information might also be referred to as an infinite-term lease for the file, with explicit lease cancellation. NQNFS, on the other hand, uses a short-term lease that expires due to

timeout after a maximum of one minute, unless explicitly renewed by the client. The fundamental difference is that an NQNFS client must keep renewing a lease to use cached data whereas a Sprite client assumes the data is valid until canceled by the server or the file is closed. Using leases permits the server to remain "stateless," since the soft state information, which consists of the set of current leases, is moot after one minute, when all the leases expire.

Whenever a client wishes to access a file's data it must hold one of three types of lease: read-caching, write-caching or non-caching. The latter type requires that all file operations be done synchronously with the server via the appropriate RPCs.

A read-caching lease allows for client data caching but no modifications may be done. It may, however, be shared between multiple clients. Diagram 1 shows a typical read-caching scenario. The vertical solid black lines depict the lease records. Note that the time lines are nowhere near to scale, since a client/server interaction will normally take less than one hundred milliseconds, whereas the normal lease duration is thirty seconds. Every lease includes a *modrev* value, which changes upon every modification of the file. It may be used to check to see if data cached on the client is still current.

A write-caching lease permits delayed write caching, but requires that all data be pushed to the server when the lease expires or is terminated by an eviction callback. When a write-caching lease has almost expired, the client will attempt to extend the lease if the file is still open, but is required to push the delayed writes to the server if renewal fails (as depicted by diagram 2). The writes may not arrive at the server until after the write lease has expired on the client, but this does not result in a consistency problem, so long as the write lease is still valid on the server. Note that, in diagram 2, the lease record on the server remains current after the expiry time, due to the conditions mentioned in section 5. If a write RPC is done on the server after the write lease has expired on the server, this could be considered an error since consistency could be lost, but it is not handled as such by NQNFS.

Diagram 3 depicts how read and write leases are replaced by a non-caching lease when there is the potential for write sharing. A write-caching lease is not used in the Stanford V Distributed System [Gray89], since synchronous writing is always used. A side effect of this change is that the five to ten second lease duration recommended by Gray was found to be insufficient to achieve good performance for the write-caching lease. Experimentation showed that thirty seconds was about optimal for cases where the client and server are connected to the same local area network, so thirty seconds is the default lease duration for NQNFS. A maximum of twice that value is permitted, since Gray showed that for some network topologies, a larger lease duration functions better. Although there is an explicit *get_lease* RPC defined for the protocol, most lease requests are piggybacked onto the other RPCs to minimize the additional overhead introduced by leasing.

4.1 Rationale

Leasing was chosen over hard server state information for the following reasons:

1. The server must maintain state information about all current client leases. Since at most one lease is allocated for each RPC and the leases expire after their lease term, the upper bound on the number of current leases is the product of the lease term and the server RPC rate. In practice, it has been observed that less than 10% of RPCs request new leases and since most leases have a term of thirty seconds, the following rule of thumb should estimate the number of server lease records:

$$\text{Number of Server Lease Records} = 0.1 * 30 * \text{RPC rate}$$

Since each lease record occupies 64 bytes of server memory, storing the lease records should not be a serious problem. If a server has exhausted lease storage, it can simply wait a few seconds for a lease to expire and free up a record. On the other hand, a Sprite-like server must store records for all files currently open by all clients, which can require significant storage for a large, heavily loaded server. In [Mogul93], it is proposed that a mechanism vaguely similar to paging could be used to deal with this for Spritely NFS, but this appears to introduce a fair amount of complexity and may limit the usefulness of open records for storing other state information, such as file locks.

2. After a server crashes it must recover lease records for the current outstanding leases, which actually implies that if it waits until all leases have expired, there is no state to recover. The server must wait for the maximum lease duration of one minute, and it must serve all outstanding write requests resulting from terminated

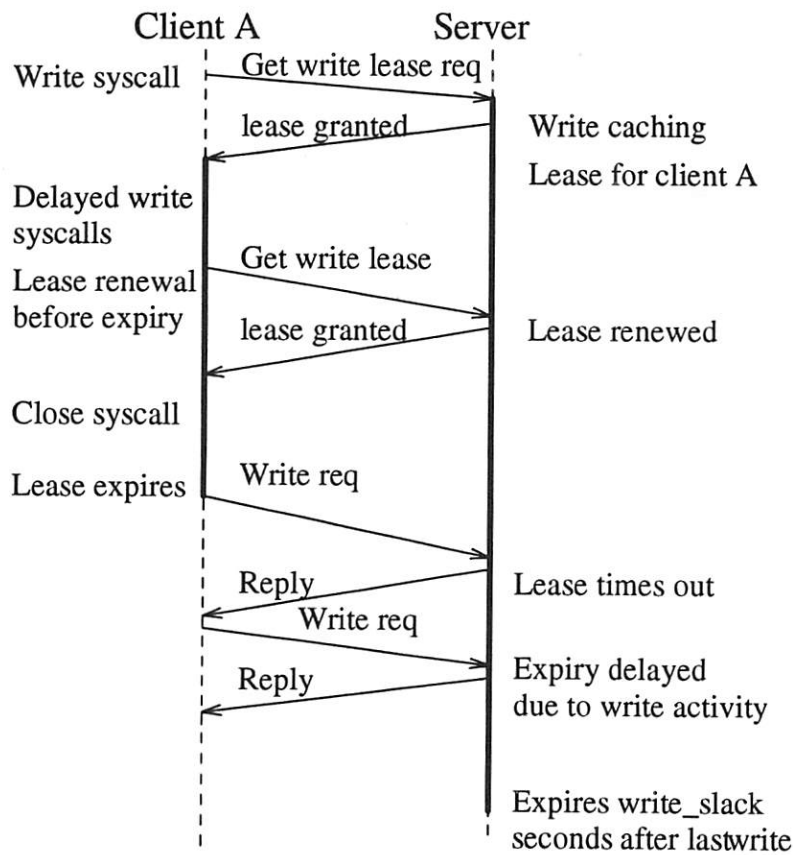


Diagram #2: Write Caching Lease

5. Limitations of the NQNFS Protocol

There is a serious risk when leasing is used for delayed write caching. If the server is simply too busy to service a lease renewal before a write-caching lease terminates, the client will not be able to push the write data to the server before the lease has terminated, resulting in inconsistency. Note that the danger of inconsistency occurs when the server assumes that a write-caching lease has terminated before the client has had the opportunity to write the data back to the server. In an effort to avoid this problem, the NQNFS server does not assume that a write-caching lease has terminated until three conditions are met:

- 1 - clock time > (expiry time + clock skew)
- 2 - there is at least one server daemon (nfsd) waiting for an RPC request
- 3 - no write RPCs received for leased file within write_slack after the corrected expiry time

The first condition ensures that the lease has expired on the client. The clock_skew, by default three seconds, must be set to a value larger than the maximum time-of-day clock error that is likely to occur during the maximum lease duration. The second condition attempts to ensure that the client is not waiting for replies to any writes that are still queued for service by an nfsd. The third condition tries to guarantee that the client has transmitted all write requests to the server, since write_slack is set to several times the client's timeout retransmit interval.

There are also certain file system semantics that are problematic for both NFS and NQNFS, due to the lack of state information maintained by the server. If a file is unlinked on one client while open on another it will be removed from the file server, resulting in failed file accesses on the client that has the file open. If the file system on the server is out of space or the client user's disk quota has been exceeded, a delayed write can fail long after the write system call was successfully completed. With NFS this error will be detected by the close system call, since the delayed writes are pushed upon close. With NQNFS however, the delayed write RPC may not occur until after the close system call, possibly even after the process has exited. Therefore, if a process must check for write

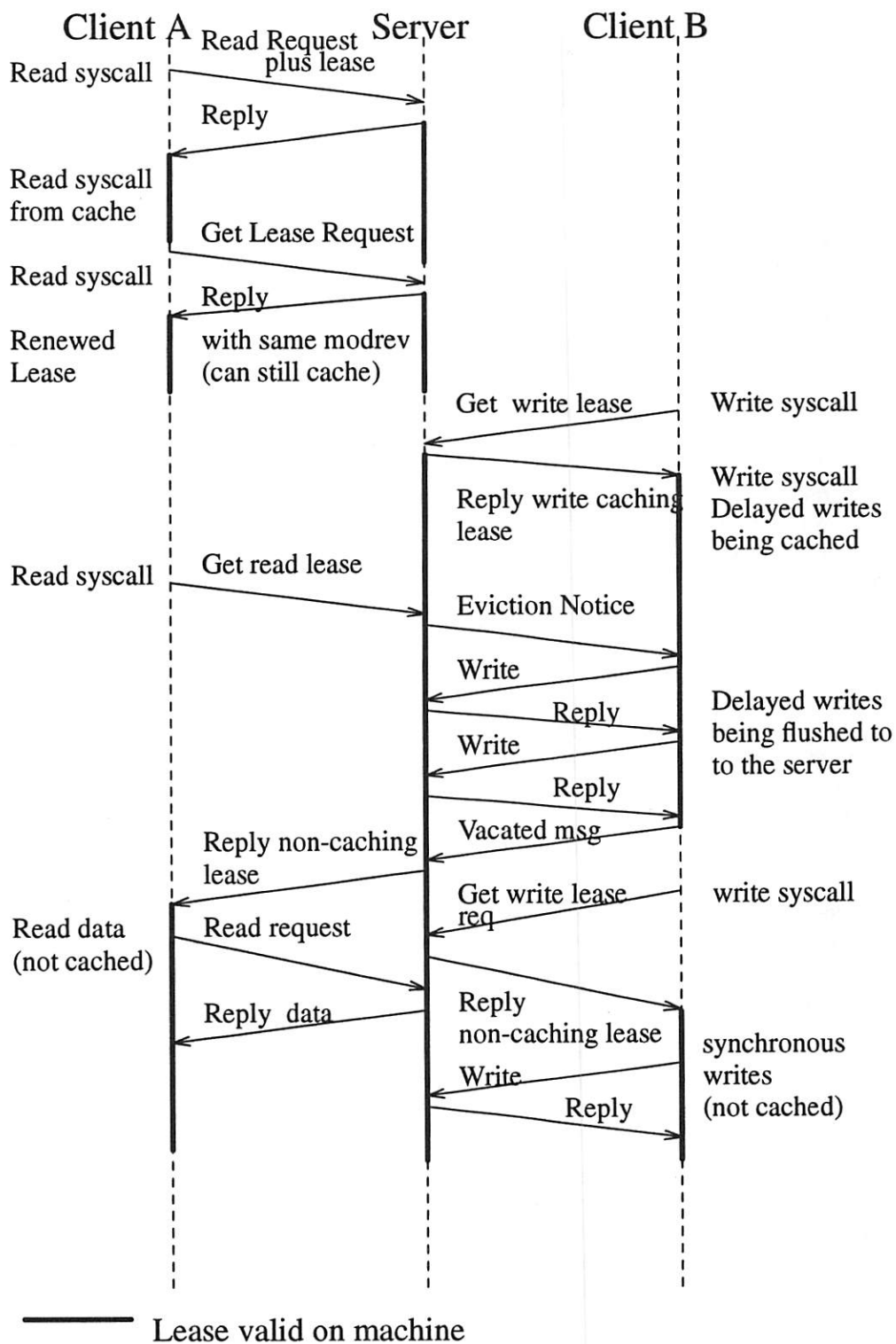


Diagram #3: Write sharing case

errors, a system call such as *fsync* must be used.

Another problem occurs when a process on one client is running an executable file and a process on another client starts to write to the file. The read lease on the first client is terminated by the server, but the client has no recourse but to terminate the process, since the process is already in progress on the old executable.

The NQNFS protocol does not support file locking, since a file lock would have to involve hard, recovered after a crash, state information.

6. Other NQNFS Protocol Features

NQNFS also includes a variety of minor modifications to the NFS protocol, in an attempt to address various limitations. The protocol uses 64bit file sizes and offsets in order to handle large files. TCP transport may be used as an alternative to UDP for cases where UDP does not perform well. Transport mechanisms such as TCP also permit the use of much larger read/write data sizes, which might improve performance in certain environments.

The NQNFS protocol replaces the *Readdir* RPC with a *Readdir_and_Lookup* RPC that returns the file handle and attributes for each file in the directory as well as name and file id number. This additional information may then be loaded into the lookup and file-attribute caches on the client. Thus, for cases such as "ls -l", the *stat* system calls can be performed locally without doing any lookup or *getattr* RPCs. Another additional RPC is the *Access* RPC that checks for file accessibility against the server. This is necessary since in some cases the client user ID is mapped to a different user on the server and doing the access check locally on the client using file attributes and client credentials is not correct. One case where this becomes necessary is when the NQNFS mount point is using Kerberos authentication, where the Kerberos authentication ticket is translated to credentials on the server that are mapped to the client side user id. For further details on the protocol, see [Macklem93].

7. Performance

In order to evaluate the effectiveness of the NQNFS protocol, a benchmark was used that was designed to typify real work on the client workstation. Benchmarks, such as Laddis [Wittle93], that perform server load characterization are not appropriate for this work, since it is primarily client caching efficiency that needs to be evaluated. Since these tests are measuring overall client system performance and not just the performance of the file system, each sequence of runs was performed on identical hardware and operating system in order to factor out the system components affecting performance other than the file system protocol.

The equipment used for all the benchmarks are members of the DECstation™† family of workstations using the MIPS™§ RISC architecture. The operating system running on these systems was a pre-release version of 4.4BSD Unix™‡. For all benchmarks, the file server was a DECstation 2100 (10 MIPS) with 8Mbytes of memory and a local RZ23 SCSI disk (27msec average access time). The clients range in speed from DECstation 2100s to a DECstation 5000/25, and always run with six block I/O daemons and a 4Mbyte buffer cache, except for the test runs where the buffer cache size was the independent variable. In all cases /tmp is mounted on the local SCSI disk², all machines were attached to the same uncongested Ethernet, and ran in single user mode during the benchmarks. Unless noted otherwise, test runs used UDP RPC transport and the results given are the average values of four runs.

The benchmark used is the Modified Andrew Benchmark (MAB) [Ousterhout90], which is a slightly modified version of the benchmark used to characterize performance of the Andrew ITC file system [Howard88]. The MAB was set up with the executable binaries in the remote mounted file system and the final load step was commented out, due to a linkage problem during testing under 4.4BSD. Therefore, these results are not directly comparable to other reported MAB results. The MAB is made up of five distinct phases:

1. Makes five directories (no significant cost)
2. Copy a file system subtree to a working directory

² Testing using the 4.4BSD MFS [McKusick90] resulted in slightly degraded performance, probably since the machines only had 16Mbytes of memory, and so paging increased.

3. Get file attributes (stat) of all the working files
4. Search for strings (grep) in the files
5. Compile a library of C sources and archive them

Of the five phases, the fifth is by far the largest and is the one affected most by client caching mechanisms. The results for phase #1 are invariant over all the caching mechanisms.

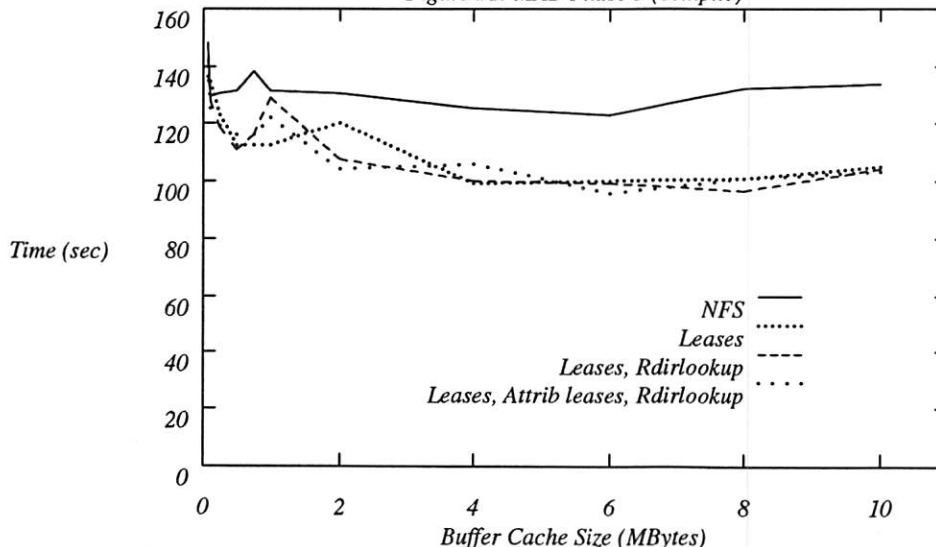
7.1 Buffer Cache Size Tests

The first experiment was done to see what effect changing the size of the buffer cache would have on client performance. A single DECstation 5000/25 was used to do a series of runs of MAB with different buffer cache sizes for four variations of the file system protocol. The four variations are as follows:

- Case 1: NFS - The NFS protocol as implemented in 4.4BSD
- Case 2: Leases - The NQNFS protocol using leases for cache consistency
- Case 3: Leases, Rdirlookup - The NQNFS protocol using leases for cache consistency and with the readdir RPC replaced by Readdir_and_Lookup
- Case 4: Leases, Attrib leases, Rdirlookup - The NQNFS protocol using leases for cache consistency, with the readdir RPC replaced by the Readdir_and_Lookup, and requiring a valid lease not only for file-data access, but also for file-attribute access.

As can be seen in figure 1, the buffer cache achieves about optimal performance for the range of two to ten megabytes in size. At eleven megabytes in size, the system pages heavily and the runs did not complete in a reasonable time. Even at 64Kbytes, the buffer cache improves performance over no buffer cache by a significant margin of 136-148 seconds versus 239 seconds. This may be due, in part, to the fact that the Compile Phase of the MAB uses a rather small working set of file data. All variants of NQNFS achieve about the same performance, running around 30% faster than NFS, with a slightly larger difference for large buffer cache sizes. Based on these results, all remaining tests were run with the buffer cache size set to 4Mbytes. Although I do not know what causes the local peak in the curves between 0.5 and 2 megabytes, there is some indication that contention for buffer cache blocks, between the update process (which pushes delayed writes to the server every thirty seconds) and the I/O system calls, may be involved.

Figure #1: MAB Phase 5 (compile)



7.2 Multiple Client Load Tests

During preliminary runs of the MAB, it was observed that the server RPC counts were reduced significantly by NQNFS as compared to NFS (table 1). (Spritely NFS and UltrixTM4.3/NFS numbers were taken from [Mogul93] and are not directly comparable, due to numerous differences in the experimental setup including deletion of the load step from phase 5.) This suggests that the NQNFS protocol might scale better with respect to the

number of clients accessing the server. The experiment described in this section ran the MAB on from one to ten clients concurrently, to observe the effects of heavier server load. The clients were started at roughly the same time by pressing all the <return> keys together and, although not synchronized beyond that point, all clients would finish the test run within about two seconds of each other. This was not a realistic load of N active clients, but it did result in a reproducible increasing client load on the server. The results for the four variants are plotted in figures 2-5.

Table #1: MAB RPC Counts							
RPC	Getattr	Read	Write	Lookup	Other	GetLease/Open-Close	Total
BSD/NQNFS	277	139	306	575	294	127	1718
BSD/NFS	1210	506	451	489	238	0	2894
Spritely NFS	259	836	192	535	306	1467	3595
Ultrix4.3/NFS	1225	1186	476	810	305	0	4002

For the MAB benchmark, the NQNFS protocol reduces the RPC counts significantly, but with a minimum of extra overhead (the GetLease/Open-Close count).

Figure #2: MAB Phase 2 (copying)

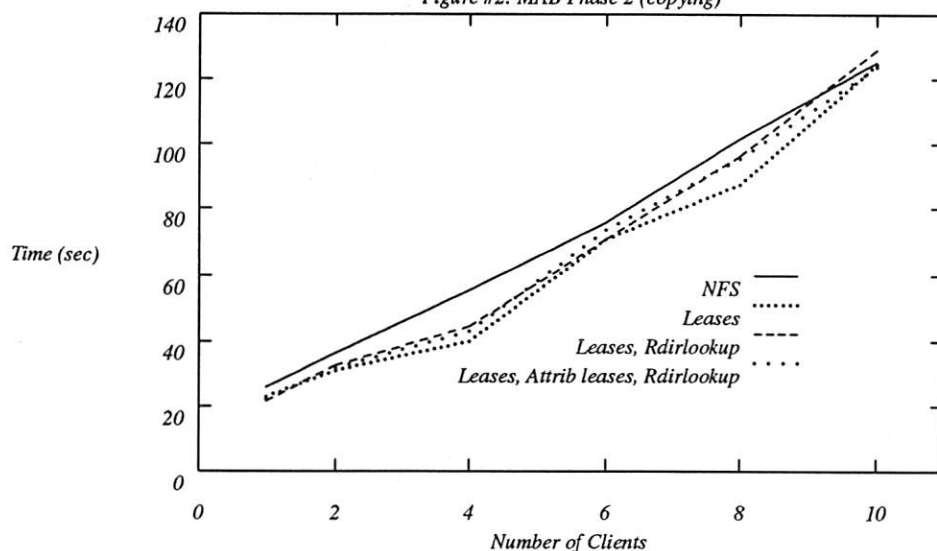
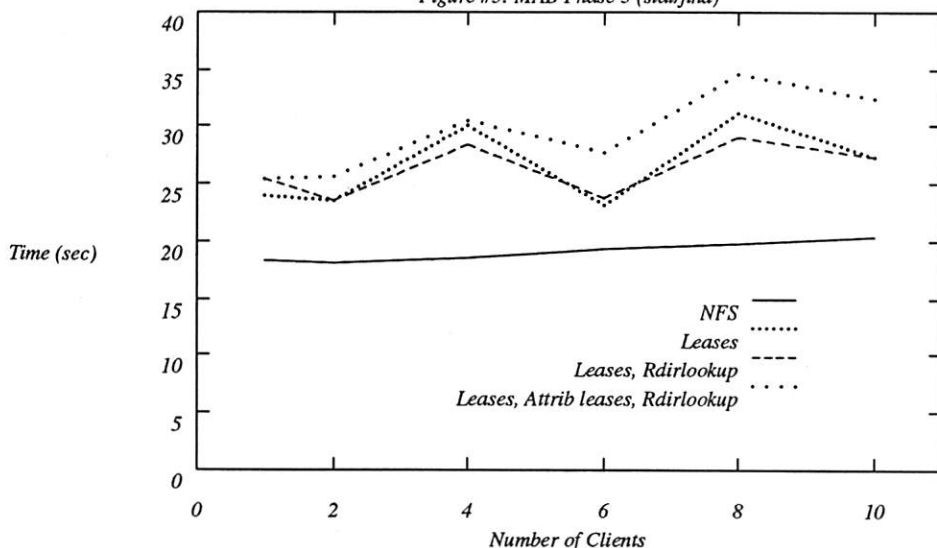
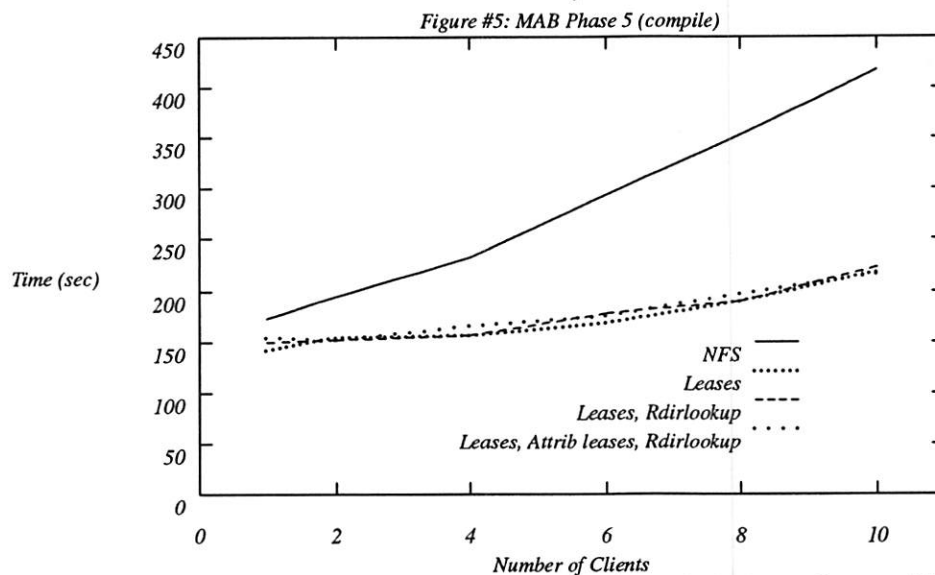
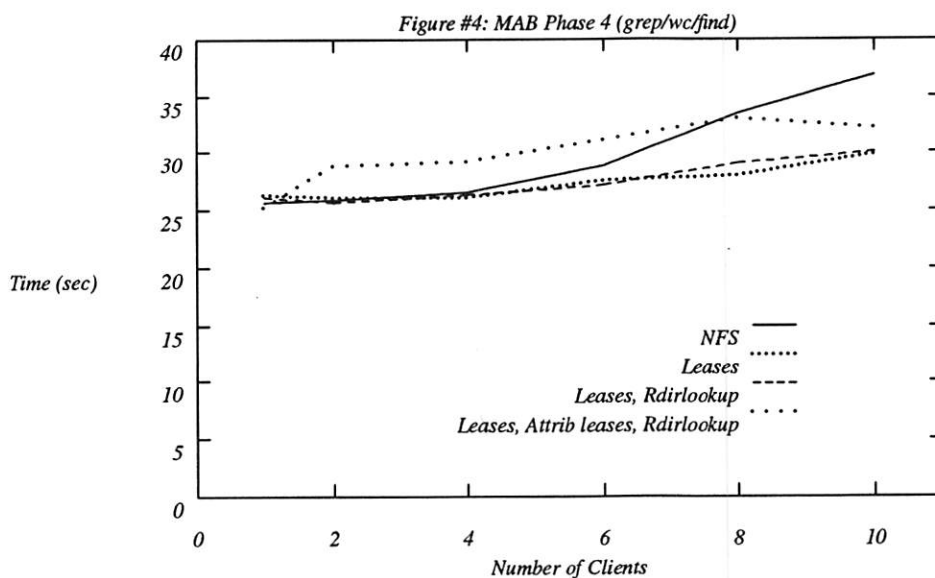


Figure #3: MAB Phase 3 (stat/find)



In figure 2, where a subtree of seventy small files is copied, the difference between the protocol variants is minimal, with the NQNFS variants performing slightly better. For this case, the Readdir_and_Lookup RPC is a



slight hindrance under heavy load, possibly because it results in larger directory blocks in the buffer cache.

In figure 3, for the phase that gets file attributes for a large number of files, the leasing variants take about 50% longer, indicating that there are performance problems in this area. For the case where valid current leases are required for every file when attributes are returned, the performance is significantly worse than when the attributes are allowed to be stale by a few seconds on the client. I have not been able to explain the oscillation in the curves for the Lease cases.

For the string searching phase depicted in figure 4, the leasing variants that do not require valid leases for files when attributes are returned appear to scale better with server load than NFS. However, the effect appears to be negligible until the server load is fairly heavy.

Most of the time in the MAB benchmark is spent in the compilation phase and this is where the differences between caching methods are most pronounced. In figure 5 it can be seen that any protocol variant using Leases performs about a factor of two better than NFS at a load of ten clients. This indicates that the use of NQNFS may allow servers to handle significantly more clients for this type of workload.

Table 2 summarizes the MAB run times for all phases for the single client DECstation 5000/25. The *Leases* case refers to using leases, whereas the *Leases, Rdir* case uses the *Readdir_and_Lookup* RPC as well and the *BCache Only* case uses leases, but only the buffer cache and not the attribute or name caches. The *No Caching* case does not do any client side caching, performing all system calls via synchronous RPCs to the server.

Table #2: Single DECstation 5000/25 Client Elapsed Times (sec)							
Phase	1	2	3	4	5	Total	% Improvement
No Caching	6	35	41	40	258	380	-93
NFS	5	24	15	20	133	197	0
BCache Only	5	20	24	23	116	188	5
Leases, Rdir	5	20	21	20	105	171	13
Leases	5	19	21	21	99	165	16

7.3 Processor Speed Tests

An important goal of client-side file system caching is to decouple the I/O system calls from the underlying distributed file system, so that the client's system performance might scale with processor speed. In order to test this, a series of MAB runs were performed on three DECstations that are similar except for processor speed. In addition to the four protocol variants used for the above tests, runs were done with the client caches turned off, for worst case performance numbers for caching mechanisms with a 100% miss rate. The CPU utilization was measured, as an indicator of how much the processor was blocking for I/O system calls. Note that since the systems were running in single user mode and otherwise quiescent, almost all CPU activity was directly related to the MAB run. The results are presented in table 3. The CPU time is simply the product of the CPU utilization and elapsed running time and, as such, is the optimistic bound on performance achievable with an ideal client caching scheme that never blocks for I/O.

	Table #3: MAB Phase 5 (compile)								
	DS2100 (10.5 MIPS)			DS3100 (14.0 MIPS)			DS5000/25 (26.7 MIPS)		
	Elapsed time	CPU Util(%)	CPU time	Elapsed time	CPU Util(%)	CPU time	Elapsed time	CPU Util(%)	CPU time
Leases	143	89	127	113	87	98	99	89	88
Leases, Rdir	150	89	134	110	91	100	105	88	92
BCache Only	169	85	144	129	78	101	116	75	87
NFS	172	77	132	135	74	100	133	71	94
No Caching	330	47	155	256	41	105	258	39	101

As can be seen in the table, any caching mechanism achieves significantly better performance than when caching is disabled, roughly doubling the CPU utilization with a corresponding reduction in run time. For NFS, the CPU utilization is dropping with increase in CPU speed, which would suggest that it is not scaling with CPU speed. For the NQNFS variants, the CPU utilization remains at just below 90%, which suggests that the caching mechanism is working well and scaling within this CPU range. Note that for this benchmark, the ratio of CPU times for the DECstation 3100 and DECstation 5000/25 are quite different than the Dhrystone MIPS ratings would suggest.

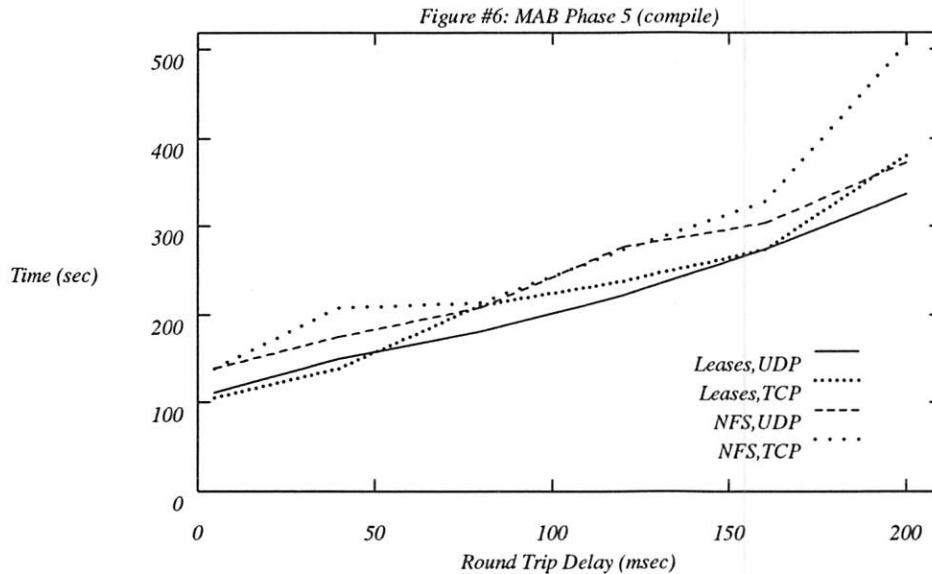
Overall, the results seem encouraging, although it remains to be seen whether or not the caching provided by NQNFS can continue to scale with CPU performance. There is a good indication that NQNFS permits a server to scale to more clients than does NFS, at least for workloads akin to the MAB compile phase. A more difficult question is "What if the server is much faster doing write RPCs?" as a result of some technology such as Prestoserve or write gathering. Since a significant part of the difference between NFS and NQNFS is the synchronous writing, it is difficult to predict how much a server capable of fast write RPCs will negate the performance improvements of NQNFS. At the very least, table 1 indicates that the write RPC load on the server has decreased by approximately 30%, and this reduced write load should still result in some improvement.

Indications are that the ReadDir_and_Lookup RPC has not improved performance for these tests and may in fact be degrading performance slightly. The results in figure 3 indicate some problems, possibly with handling of the attribute cache. It seems logical that the ReadDir_and_Lookup RPC should be permit priming of the attribute cache improving hit rate, but the results are counter to that.

7.4 Internetwork Delay Tests

This experimental setup was used to explore how the different protocol variants might perform over internetworks with larger RPC RTTs. The server was moved to a separate Ethernet, using a MicroVAXII™ as an IP router to the other Ethernet. The 4.3Reno BSD Unix system running on the MicroVAXII was modified to delay IP

packets being forwarded by a tunable N millisecond delay. The implementation was rather crude and did not try to simulate a distribution of delay times nor was it programmed to drop packets at a given rate, but it served as a simple emulation of a long, fat network³ [Jacobson88]. The MAB was run using both UDP and TCP RPC transports for a variety of RTT delays from five to two hundred milliseconds, to observe the effects of RTT delay on RPC transport. It was found that, due to a high variability between runs, four runs was not suffice, so eight runs at each value was done. The results in figure 6 and table 4 are the average for the eight runs.



Delay (msec)	NFS,UDP		NFS,TCP		Leases,UDP		Leases,TCP	
	Elapsed time (sec)	Standard Deviation	Elapsed time (sec)	Standard Deviation	Elapsed time (sec)	Standard Deviation	Elapsed time (sec)	Standard Deviation
5	139	2.9	139	2.4	112	7.0	108	6.0
40	175	5.1	208	44.5	150	23.8	139	4.3
80	207	3.9	213	4.7	180	7.7	210	52.9
120	276	29.3	273	17.1	221	7.7	238	5.8
160	304	7.2	328	77.1	275	21.5	274	10.1
200	372	35.0	506	235.1	338	25.2	379	69.2

I found these results somewhat surprising, since I had assumed that stability across an internetwork connection would be a function of RPC transport protocol. Looking at the standard deviations observed between the eight runs, there is an indication that the NQNFS protocol plays a larger role in maintaining stability than the underlying RPC transport protocol. It appears that NFS over TCP transport is the least stable variant tested. It should be noted that the TCP implementation used was roughly at 4.3BSD Tahoe release and that the 4.4BSD TCP implementation was far less stable and would fail intermittently, due to a bug I was not able to isolate. It would appear that some of the recent enhancements to the 4.4BSD TCP implementation have a detrimental effect on the performance of RPC-type traffic loads, which intermix small and large data transfers in both directions. It is obvious that more exploration of this area is needed before any conclusions can be made beyond the fact that over a local area network, TCP transport provides performance comparable to UDP.

8. Lessons Learned

Evaluating the performance of a distributed file system is fraught with difficulties, due to the many software and hardware factors involved. The limited benchmarking presented here took a considerable amount of time and the results gained by the exercise only give indications of what the performance might be for a few scenarios.

³ Long fat networks refer to network interconnections with a Bandwidth X RTT product $> 10^5$ bits.

The IP router with delay introduction proved to be a valuable tool for protocol debugging⁴, and may be useful for a more extensive study of performance over internetworks if enhanced to do a better job of simulating internetwork delay and packet loss.

The Leases mechanism provided a simple model for the provision of cache consistency and did seem to improve performance for various scenarios. Unfortunately, it does not provide the server state information that is required for file system semantics, such as locking, that many software systems demand. In production environments on my campus, the need for file locking and the correct generation of the ETXTBSY error code are far more important than full cache consistency, and leasing does not satisfy these needs. Another file system semantic that requires hard server state is the delay of file removal until the last close system call. Although Spritely NFS did not support this semantic either, it is logical that the open file state maintained by that system would facilitate the implementation of this semantic more easily than would the Leases mechanism.

9. Further Work

The current implementation uses a fixed, moderate sized buffer cache designed for the local UFS [McKusick84] file system. The results in figure 1 suggest that this is adequate so long as the cache is of an appropriate size. However, a mechanism permitting the cache to vary in size has been shown to outperform fixed sized buffer caches [Nelson90], and could be beneficial. It could also be useful to allow the buffer cache to grow very large by making use of local backing store for cases where server performance is limited. A very large buffer cache size would in turn permit experimentation with much larger read/write data sizes, facilitating bulk data transfers across long fat networks, such as will characterize the Internet of the near future. A careful redesign of the buffer cache mechanism to provide support for these features would probably be the next implementation step.

The results in figure 3 indicate that the mechanics of caching file attributes and maintaining the attribute cache's consistency needs to be looked at further. There also needs to be more work done on the interaction between a Readdir_and_Lookup RPC and the name and attribute caches, in an effort to reduce Getattr and Lookup RPC loads.

The NQNFS protocol has never been used in a production environment and doing so would provide needed insight into how well the protocol satisfies the needs of real workstation environments. It is hoped that the distribution of the implementation in 4.4BSD will facilitate use of the protocol in production environments elsewhere.

The big question that needs to be resolved is whether Leases are an adequate mechanism for cache consistency or whether hard server state is required. Given the work presented here and in the papers related to Sprite and Spritely NFS, there are clear indications that a cache consistency algorithm can improve both performance and file system semantics. As yet, however, it is unclear what the best approach to maintain consistency is. It would appear that hard state information is required for file locking and other mechanisms and, if so, it seems appropriate to use it for cache consistency as well.

10. Acknowledgements

I would like to thank the members of the CSRG at the University of California, Berkeley for their continued support over the years. Without their encouragement and assistance this software would never have been implemented. Prof. Jim Linders and Prof. Tom Wilson here at the University of Guelph helped proofread this paper and Jeffrey Mogul provided a great deal of assistance, helping to turn my gibberish into something at least moderately readable.

11. References

- [Baker91] Mary Baker and John Ousterhout, Availability in the Sprite Distributed File System, In *Operating System Review*, (25)2, pg. 95-98, April 1991.
- [Baker91a] Mary Baker, private communication, May 1991.

⁴ It exposed two bugs in the 4.4BSD networking, one a problem in the Lance chip driver for the DECstation and the other a TCP window sizing problem that I was not able to isolate.

- [Burrows88] Michael Burrows, Efficient Data Sharing, Technical Report #153, Computer Laboratory, University of Cambridge, Dec. 1988.
- [Gray89] Cary G. Gray and David R. Cheriton, Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency, In *Proc. of the Twelfth ACM Symposium on Operating Systems Principals*, Litchfield Park, AZ, Dec. 1989.
- [Howard88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham and Michael J. West, Scale and Performance in a Distributed File System, *ACM Trans. on Computer Systems*, (6)1, pg 51-81, Feb. 1988.
- [Jacobson88] Van Jacobson and R. Braden, *TCP Extensions for Long-Delay Paths*, ARPANET Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA, October 1988, RFC-1072.
- [Jacobson89] Van Jacobson, Sun NFS Performance Problems, *Private Communication*, November, 1989.
- [Juszczak89] Chet Juszczak, Improving the Performance and Correctness of an NFS Server, In *Proc. Winter 1989 USENIX Conference*, pg. 53-63, San Diego, CA, January 1989.
- [Juszczak94] Chet Juszczak, Improving the Write Performance of an NFS Server, to appear in *Proc. Winter 1994 USENIX Conference*, San Francisco, CA, January 1994.
- [Kazar88] Michael L. Kazar, Synchronization and Caching Issues in the Andrew File System, In *Proc. Winter 1988 USENIX Conference*, pg. 27-36, Dallas, TX, February 1988.
- [Kent87] Christopher. A. Kent and Jeffrey C. Mogul, *Fragmentation Considered Harmful*, Research Report 87/3, Digital Equipment Corporation Western Research Laboratory, Dec. 1987.
- [Kent87a] Christopher. A. Kent, *Cache Coherence in Distributed Systems*, Research Report 87/4, Digital Equipment Corporation Western Research Laboratory, April 1987.
- [Macklem90] Rick Macklem, Lessons Learned Tuning the 4.3BSD Reno Implementation of the NFS Protocol, In *Proc. Winter 1991 USENIX Conference*, pg. 53-64, Dallas, TX, January 1991.
- [Macklem93] Rick Macklem, The 4.4BSD NFS Implementation, In *The System Manager's Manual*, 4.4 Berkeley Software Distribution, University of California, Berkeley, June 1993.
- [McKusick84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler and Robert S. Fabry, A Fast File System for UNIX, *ACM Transactions on Computer Systems*, Vol. 2, Number 3, pg. 181-197, August 1984.
- [McKusick90] Marshall K. McKusick, Michael J. Karels and Keith Bostic, A Pageable Memory Based Filesystem, In *Proc. Summer 1990 USENIX Conference*, pg. 137-143, Anaheim, CA, June 1990.
- [Mogul93] Jeffrey C. Mogul, Recovery in Spritely NFS, Research Report 93/2, Digital Equipment Corporation Western Research Laboratory, June 1993.
- [Moran90] Joseph Moran, Russel Sandberg, Don Coleman, Jonathan Kepecs and Bob Lyon, Breaking Through the NFS Performance Barrier, In *Proc. Spring 1990 EUUG Conference*, pg. 199-206, Munich, FRG, April 1990.
- [Nelson88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout, Caching in the Sprite Network File System, *ACM Transactions on Computer Systems* (6)1 pg. 134-154, February 1988.
- [Nelson90] Michael N. Nelson, *Virtual Memory vs. The File System*, Research Report 90/4, Digital Equipment Corporation Western Research Laboratory, March 1990.
- [Nowicki89] Bill Nowicki, Transport Issues in the Network File System, In *Computer Communication Review*, pg. 16-20, March 1989.
- [Ousterhout90] John K. Ousterhout, Why Aren't Operating Systems Getting Faster As Fast as Hardware? In *Proc. Summer 1990 USENIX Conference*, pg. 247-256, Anaheim, CA, June 1990.
- [Sandberg85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon, Design and Implementation of the Sun Network filesystem, In *Proc. Summer 1985 USENIX Conference*, pages 119-130, Portland, OR, June 1985.

- [Srinivasan89] V. Srinivasan and Jeffrey. C. Mogul, Spritely NFS: Experiments with Cache-Consistency Protocols, In *Proc. of the Twelfth ACM Symposium on Operating Systems Principles*, Litchfield Park, AZ, Dec. 1989.
- [Steiner88] J. G. Steiner, B. C. Neuman and J. I. Schiller, Kerberos: An Authentication Service for Open Network Systems, In *Proc. Winter 1988 USENIX Conference*, pg. 191-202, Dallas, TX, February 1988.
- [SUN89] Sun Microsystems Inc., *NFS: Network File System Protocol Specification*, ARPANET Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA, March 1989, RFC-1094.
- [SUN93] Sun Microsystems Inc., *NFS: Network File System Version 3 Protocol Specification*, Sun Microsystems Inc., Mountain View, CA, June 1993.
- [Wittle93] Mark Wittle and Bruce E. Keith, LADDIS: The Next Generation in NFS File Server Benchmarking, In *Proc. Summer 1993 USENIX Conference*, pg. 111-128, Cincinnati, OH, June 1993.

Appendix: Protocol Details

The protocol specification is identical to that of NFS [Sun89] except for the following changes:

RPC Information: Program Number 300105, Version 1

readdirlookres

NQNFSPROC_READDIRLOOK(fhandle file; nfscookie cookie; unsigned cnt; unsigned duration) = 18;

union readdirlookres switch (stat status) {

case NFS_OK: struct {
 entry *entries; bool eof;
 } readdirlookok;

default: void;

};

struct entry {

 unsigned cachable, duration; modifyrev rev; fhandle entry_fh;
 nqnfs_fattr entry_attr; unsigned fid; filename name; nfscookie cookie;
 entry *nextentry;

};

Reads entries in a directory in a manner analogous to the NFSPROC_READDIR RPC in NFS, but returns the file handle and attributes of each entry as well.

getleaseres NQNFSPROC_GETLEASE(fhandle file; cachetype rw; unsigned duration) = 19;

union getleaseres switch (stat status) {

case NFS_OK: bool cachable; unsigned duration; modifyrev rev; nqnfs_fattr attr;
 default: void;

};

Gets a lease for "file" valid for "duration" seconds from when the lease was issued on the server.

void NQNFSPROC_EVICTED (fhandle file) = 21;

This message is sent from the server to the client. When the client receives the message, it should flush data associated with the file represented by "fhandle" from its caches and then send the **Vacated Message** back to the server. Flushing includes pushing any dirty writes via write RPCs.

void NQNFSPROC_VACATED (fhandle file) = 20;

This message is sent from the client to the server in response to the **Eviction Message**. See above.

stat NQNFSPROC_ACCESS(fhandle file; bool read; bool write; bool exec) = 22;

The access RPC does permission checking on the server for the given type of access required by the client for the file.

The piggybacked get lease request is functionally equivalent to the Get Lease RPC except that is attached to one of the other NQNFS RPC requests as follows. A getleaserequest is prepended to all of the request arguments for NQNFS and a getleaserequestres is inserted in all NFS result structures just after the "stat" field only if "stat ==

NFS_OK".

```
union getleaserequest switch (cachetype type) {
    case NQLREAD: case NQLWRITE: unsigned duration;
    default: void;
};
union getleaserequestres switch (cachetype type) {
    case NQLREAD: case NQLWRITE: bool cachable; unsigned duration; modifyrev rev;
    default: void;
};
```

The get lease request applies to the file that the attached RPC operates on and the file attributes remain in the same location as for the NFS RPC reply structure.

Data Structures

Three additional values have been added to the enumerated type "stat".

NQNFS_EXPIRED=500, NQNFS_TRYLATER=501, NQNFS_AUTHERR=502

```
enum cachetype {
    NQLNONE = 0, NQLREAD = 1, NQLWRITE = 2
};
```

Type of lease requested. NQLNONE is used to indicate no piggybacked lease request.

```
typedef unsigned hyper modifyrev;
```

The "modifyrev" is a unsigned quadword integer value that is never zero and increases every time the corresponding file is modified on the server.

```
struct nqnfs_time {
    unsigned seconds, nano_seconds;
};
```

```
struct nqnfs_fattr {
    ftype type; unsigned mode, nlink, uid, gid; unsigned hyper size;
    unsigned blocksize, rdev; unsigned hyper bytes; unsigned fsid, fid;
    nqnfs_time atime, mtime, ctime; unsigned flags, gen; modifyrev rev;
};
```

```
struct nqnfs_sattr {
    unsigned mode, uid, gid; unsigned hyper size; nqnfs_time atime, mtime; unsigned flags, rev;
};
```

The attribute structures are modified from the NFS ones so that they store the file size as a 64bit quantity and the storage occupied as a 64bit number of bytes. They also have fields added for the 4.4BSD va_flags and va_gen as well as the file's modify rev level.

The arguments to several of the NFS RPCs have been modified as well. Mostly, these are minor changes to use 64bit file offsets or similar. The modified argument structures follow.

```
struct lookup_diropargs {
    unsigned duration; fhandle dir; filename name;
};
union lookup_diropres switch (stat status) {
case NFS_OK: struct {
    union getleaserequestres lookup_lease; fhandle file; nqnfs_fattr attr;
    } lookup_diropok;
default: void;
};
```

The additional "duration" argument tells the server to get a lease for the name being looked up if it is non-zero and the lease is specified in "lookup_lease".

```
struct nqnfs_readargs {
    fhandle file; unsigned hyper offset; unsigned count;
```

```

};
struct nqnfs_writeargs {
    fhandle file; unsigned hyper offset; bool append; nfsdata data;
};
The "append" argument is true for append only write operations.
union nqnfs_statfsres (stat status) {
case NFS_OK: struct {
    unsigned tsize, bsize, blocks, bfree, bavail, files, files_free;
    } info;
default: void;
};

```

The "files" field is the number of files in the file system and the "files_free" is the number of additional files that can be created.

Rick is a support technician with the Department of Computing and Information Science at the University of Guelph. He provides operational support for a network of around 90 Novell PCs and 35 Unix systems of various flavours. He has been working with Unix kernels since 1979, including assorted ports and the NFS related work reported in this paper and still uses *ed* (really!). His spare time interests include horses, skiing and canoeing. Rick's electronic mail address is rick@snowwhite.cis.uoguelph.ca.

† Prestoserve is a trademark of Legato Systems, Inc.

§ MIPS is a trademark of Silicon Graphics, Inc.

† DECstation, MicroVAXII and Ultrix are trademarks of Digital Equipment Corp.

‡ Unix is a trademark of Novell, Inc.

A Quantitative Analysis of Disk Drive Power Management in Portable Computers

Kester Li, Roger Kumpf, Paul Horton, and Thomas Anderson

*Computer Science Division
University of California
Berkeley, CA 94720*

Abstract

With the advent and subsequent popularity of portable computers, power management of system components has become an important issue. Current portable computers implement a number of power reduction techniques to achieve a longer battery life. Included among these is spinning down a disk during long periods of inactivity. In this paper, we perform a quantitative analysis of the potential costs and benefits of spinning down the disk drive as a power reduction technique. Our conclusion is that almost all the energy consumed by a disk drive can be eliminated with little loss in performance. Although on current hardware, reliability can be impacted by our policies, the next generation of disk drives will use technology (such as dynamic head loading) which is virtually unaffected by repeated spinups. We found that the optimal spindown delay time, the amount of time the disk idles before it is spun down, is 2 seconds. This differs significantly from the 3-5 minutes in current practice by industry. We will show in this paper the effect of varying the spindown delay on power consumption; one conclusion is that a 3-5 minute delay results in only half of the potential benefit of spinning down a disk.

1 Introduction

Power management has become an important consideration in the design of new hardware and software. Portable computers today can only function for several hours before draining their battery source. Industry has taken the approach of course-grained shutdown of system components as the major power management technique. This approach works well when there are clear periods of system inactivity, but fails under more typical scattered activity patterns. We believe that opportunities exist for fine-grained power management in the portable computer environment.

Tackling the question of power management begins with an analysis of where the energy is being consumed. Table 1 gives a listing of the major system components and their power consumption in a typical portable computer. At 68%, the display clearly dominates the system power consumption. However, we did not target the display in this study because the hardware technology in that area is still rapidly evolving (it is not clear that the display will continue to be the dominant power cost). Moreover, proposed techniques to better manage the power consumption of the display (for instance, back lighting only the portion of the screen containing the cursor) would require extensive hardware changes to be practical. Instead, we focused on managing the disk drive which represents 20% of the power consumption. The disk is a promising candidate for power management because it is a device with which the user does not interact with directly. With proper management by the operating system, the disk may be spun up and down without the user noticing much difference in performance or reliability. From anecdotal observations of disk activity on personal computers, we believed that almost all the power consumed by a disk drive could be eliminated.

Component	Manufacturer & Model	Power (watts)	Percent of Total
Display	Compaq monochrome lite25c	3.5	68%
Disk Drive (105 Mbytes)	Maxtor MXL-105 III	1.0	20%
CPU	3.3V Intel486	0.6	12%
Memory (16 Mbytes)	Micron MT4C4M4A1/B1	0.024	0.5%

Table 1: Breakdown of power consumption by components.

To investigate these issues, we collected traces of file system activity from both personal computers and Unix workstations. We then simulated the effect on power consumption of different disk management strategies for our traces. Among the questions we examined were:

- How long should the disk remain idle after servicing a request before spinning down?
- Can a small, fixed spindown delay approach the optimal energy savings?
- What is the effect of spinning down the disk on the system performance observed by the user?
- What is the effect on power consumption of adding memory as a disk cache?
- Does delaying disk writes to occur in the background save power?
- Are name-attribute caches helpful for reducing power consumption?

The remainder of the paper discusses these issues in more detail. Section 2 gives background on disk power consumption and the assumptions we made in building our simulator. Section 3 outlines how we collected our file traces. Section 4 presents the results of our simulation study, and Section 5 summarizes our conclusions.

2 Background/Simulator Components

2.1 Disk

The recent explosion in the portable computer market has enticed disk drive manufacturers to develop a special breed of drives especially designed for the portable environment. In addition to high shock tolerances, reduced physical volume, and smaller weights, these drives consume less energy and more importantly have a new mode of operation called SLEEP mode. A very significant portion of the energy consumed by a disk drive is spent in preserving the angular momentum of the physical disk platter. A much smaller fraction is spent in powering the electrical components of the drive. By sleeping, a drive can reduce its energy consumption to near zero by allowing the disk platter to spin down to a resting state. This substantial energy reduction is not without its costs. An access to the disk while it is sleeping incurs a delay measured in seconds as opposed to the tens of milliseconds required for an access to a spinning disk.

The disk we simulated is a prototypical next generation low power drive heavily optimized for the portable environment. It has four major modes of operation. OFF mode is when the disk consumes no energy and is incapable of performing any functions except powerup. SLEEP mode is when the disk is powered up but the physical disk platter is not spinning. IDLE mode is characterized by a spinning disk, but the absence of disk activity. A drive attached to a personal computer typically resides in this mode. The last mode of operation is ACTIVE, when the disk platter is spinning and either the disk head is seeking or the disk head is actively reading or writing the disk. This mode consumes the most power, but occurs only for short periods of time in a typical single-user system. Figure 1 shows a state transition diagram for our simulated drive.

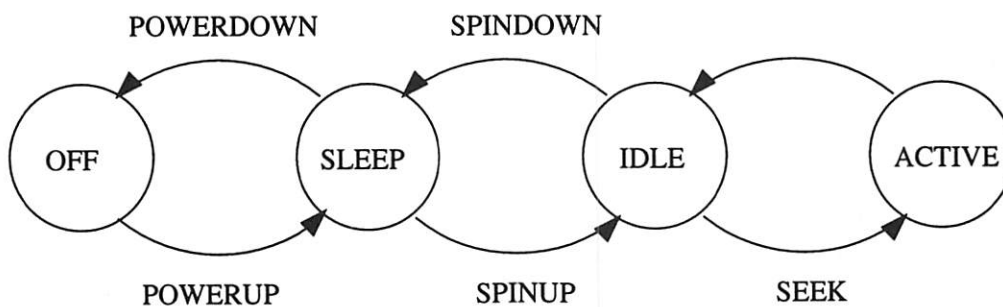


Figure 1: Disk State Transition Diagram.

Mode	Power (watts)
OFF	0.0
SLEEP	0.025
IDLE	1.0
ACTIVE	1.95

Table 2: Power consumption of the major disk modes for the Maxtor MXL-105 III.

Transition	Time (seconds)	Power (watts)
POWERUP	0.5	0.025
SPINUP	2.0	3.0
SEEK	0.009	1.95
SPINDOWN	1.0	0.025
POWERDOWN	0.5	N/A

Table 3: Average transition times between major disk modes and their power consumptions for the Maxtor MXL-105 III.

The diagram shows each of the states and the relationships between them. The disk mode transitions which require a non-zero time delay have been labeled for reference. Tables 2 and 3 quantify the power consumption of each mode and the transition times between modes. These numbers are taken from the Maxtor MXL-105 III disk drive [Maxtor].

2.2 Disk Cache

Traditionally, disk caches have been used to increase the performance of file systems. Due to the principle of locality and the large speed differences between DRAM's and disks, the operating system can use a disk cache to give the illusion of a faster disk drive. This performance boost takes on even more importance in the portable computer environment where disks sleep and the speed separation between DRAM's and disks grows even larger.

Performance, however, is not the only reason for including a disk cache in a low power operating system. From a power management viewpoint, disk operations are not only measured in terms of access times, but also in terms of energy costs. Disk operations which are requested while the disk is sleeping will incur a high energy cost to spinup the disk. If the disk cache can satisfy most of these requests, the result will be much smaller energy costs than in a system with no cache. Thus disk caches can reduce energy consumption by filtering read traffic.

The write policy of the disk cache also has impact on the design of low power operating systems. We investigate the impact of a write-back policy for our disk cache because it can make writes asynchronous with user activity, and because disk caches on portables are battery-backed non-volatile RAM. Asynchronous writes yield a measurable speed advantage in distributed and timeshared systems by overlapping computation and I/O. In addition, this policy allows writes to files that are quickly overwritten to be eliminated. More generally, asynchronous writes make sense whenever there is a high probability of long latency disk access as would occur if a disk is sleeping in a low power system. By making writes asynchronous, we can continue processing while the disk spins up for handling the write request. We thereby reduce the overall performance impact of spinning down a disk. In addition, caching writes in memory for short periods of time aids in reducing the number of spinups necessary to service writes.

Our disk cache is a typical LRU block cache with a block size of 4 kilobytes. The cache is parameterized over both the number of blocks and the write delay. A larger cache can be specified by increasing the number of blocks allocated. Greater reliability can be achieved by decreasing the write delay of the cache. The write delay specifies the maximum amount of time a dirty block may spend in the cache. A write delay of zero instantiates a write-through cache.

2.3 Name-Attribute Cache

Studying prior measurements of file system behavior, we noticed that a significant portion of file system activity is involved with reading, translating, and listing the names and attributes of files in the file system. With the goal of minimizing both energy consumption and user delay, we investigate the quantitative impact of incorporating both a name and an attribute cache in a low power system. A recent study into the effectiveness of name and attribute caches was done in [SO92]. They concluded that a twenty-directory name cache had a 97% hit rate with 2% capacity and 1% compulsory misses and that a twenty-entry attribute cache had an 88% hit rate with 5% capacity and 7% compulsory misses. Although these numbers are very good for distributed and timesharing systems, a low power system differs in that it has a much higher miss latency penalty. If a significant percent of the misses occur while the disk is sleeping, then a low power system would pay a large user delay penalty in addition to increased energy consumption. With such high costs, it seems reasonable to investigate how to further reduce the attribute cache miss rate to something near the name cache miss rate. We propose that all files in directories cached in the name cache should also have their attributes cached in memory.

Since we are caching all attributes of files in the name cache, there is no longer any need to distinguish between name and attribute caches. In our system, they have been combined into one cache which we will call the name-attribute cache. In the [SO92] study, 93% of directories had 25 or fewer files. A quick "back of the envelope" calculation shows that with 25 files per directory and 100 bytes of name, attribute, and structure overhead per file, a 20-directory cache consumes less than fifty kilobytes, which is a tiny amount of memory by today's standards.

A rudimentary study of the number of directories referenced in the DOS traces over a four hour period indicates that the number of active directories is well below twenty. With almost all cache misses due to compulsory misses, we assume in this study that the name-attribute cache pays an initial warmup penalty to read in the active directories and then performs perfectly thereafter.

3 Traces

We examine file system traces from two separate platforms. Our measurements of Microsoft DOS file system activity represent most of the applications run on portable computers today. However, in the near future, we expect to see a growing number of Unix-like applications running on increasingly powerful portables. Thus, for completeness, we also examine file system access patterns on a Unix-like platform and compare the results with the DOS traces. To be representative of portable usage with limited battery lifetimes, we used 1 hour and 4 hour long traces.

3.1 DOS Traces

A large portion of the DOS traces were taken from a student and faculty computing facility at City College of San Francisco (CCSF). All of the CCSF traces were running one of five programs: Word Perfect, Lotus 123, Quattro Pro, Paradox, and Question & Answer. These traces were taken over a period of a month and include trace lengths spanning the spectrum from thirty minutes to six hours. To further increase the representativeness of our DOS traces, we had several other trace sources running Windows 3.1, Microsoft Word, Microsoft Excel, Equation Editor, Quicken, and various text editors. These traces were taken over a period of three months. We used 61 one hour DOS traces and 15 four hour DOS traces.

3.2 Sprite Traces

For Unix-like file activity, we extracted segments from trace data collected on the Sprite distributed file system [BHK91]. These traces primarily capture the file activity of Unix programs. The only perceptible difference in the file activity of the programs running on Sprite rather than a Unix file system is that Sprite performs additional name lookups before accessing files. The traces were taken from two different weeks, one in January of 1991 and one in May of 1991. We extracted traces from three different times of the day and from twelve distinct host machines. Each trace was filtered for file system activity due solely to that machine. We used 8 one hour Sprite traces and 7 four hour Sprite traces.

The Sprite traces we used contain information about all open, lseek, and close operations, but no read or write operations except those to concurrently write-shared files. Thus, it was usually necessary to infer read and write operations from changes in the file pointer value reported for lseek and close operations. All file pointer differences in files opened for write or read/write access were attributed to unrecorded write operations. Likewise, file pointer differences in files opened for read access were attributed to unrecorded read operations. This policy resulted in approximately two thirds as many write operations as reads in our traces. We believe that inferring these read and write operations had minimal impact on the integrity of the traces, since files are generally open for very short periods of time. Measurements of the Sprite file system [BHK91] showed that over 90% of the files were open for less than 1/2 second. The traces gave us enough information to infer the number of bytes read or written, the offset of those bytes in the file, and when the operations occurred to within a tolerance of about 1/4 second. It should be noted that the remaining 10% of the files have open-close times which are distributed roughly exponentially. In this paper, we assume that the disk operations for these files occur at close time.

A complete analysis of disk activity would rightfully include the paging activity of the system. Unfortunately for our study, the Sprite traces we examined did not contain paging data and we could not account for these disk references. However, for performance reasons, we expect that the amount of disk activity due to paging is small compared to file system activity.

4 Evaluation

We split the DOS and Sprite traces into one hour and four hour lengths to analyze any differences in reference patterns. Our one hour results were consistent with the four hour results so we present only the latter set in the following sections. The DOS traces were grouped according to applications and analyzed using the same criteria. There were some minor differences in energy consumption characteristics between software vendors, but no notable differences along application domains such as databases or spreadsheets. This result is somewhat surprising and is most likely due to differences in run-time memory management systems. As the differences were minor, we again decided to suppress this distinction by averaging the results.

4.1 Spindown Delay

4.1.1 Spindown Delay Curve

Figures 2 and 3 show the energy consumption of the disk as a function of spindown delay – that is, the length of time we wait for further disk activity before allowing the disk to stop rotating. Figure 3 shows

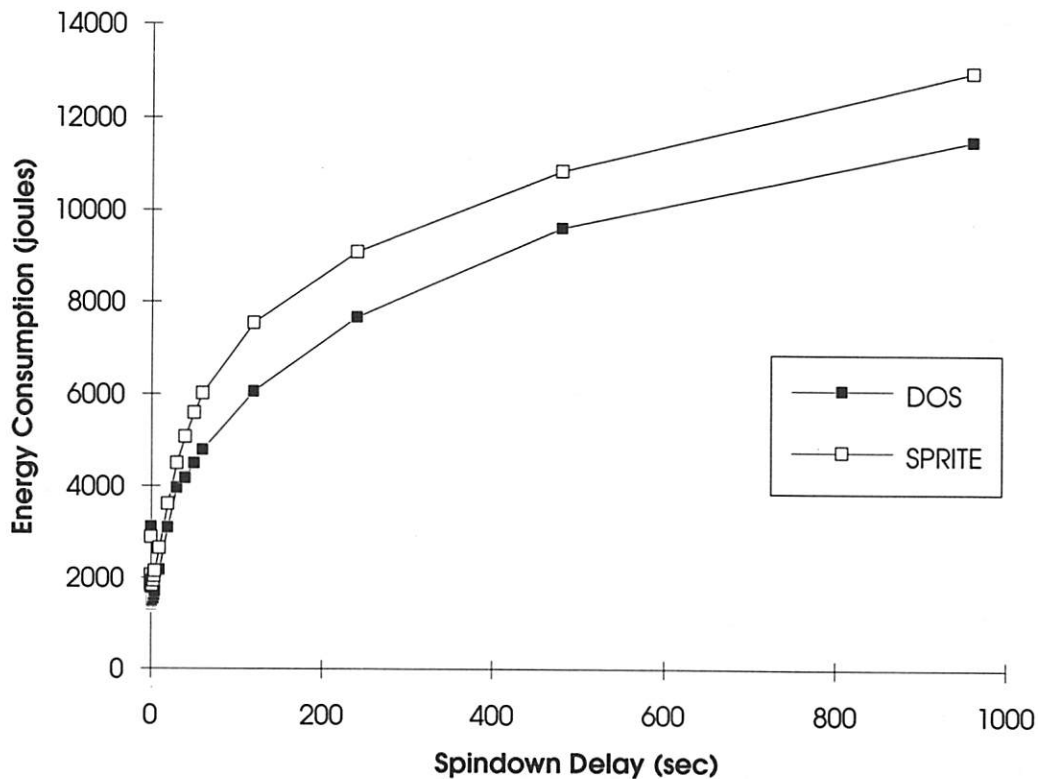


Figure 2: **Spindown Delay vs. Energy Consumption.** This figure shows the effect of varying spindown delay on energy consumption. The simulations were run with a one megabyte disk cache, 30 second write delay, name-attribute caching enabled and 4 hour traces.

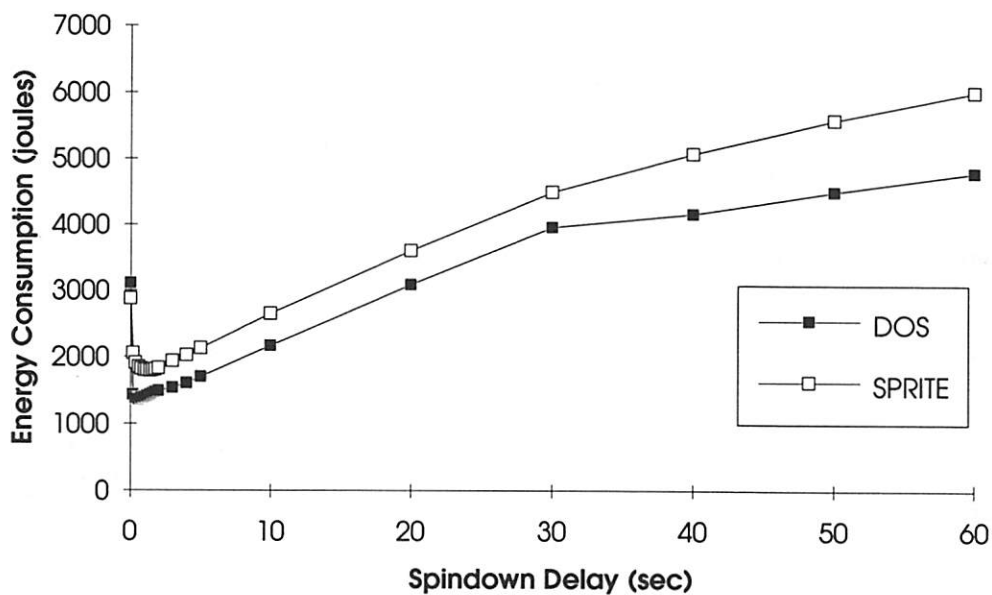


Figure 3: **Spindown Delay vs. Energy Consumption (Minimal Region).** This figure shows the effect of varying spindown delay on energy consumption for small spindown delay values. The simulations were run with a one megabyte disk cache, 30 second write delay, name-attribute caching enabled and 4 hour traces.

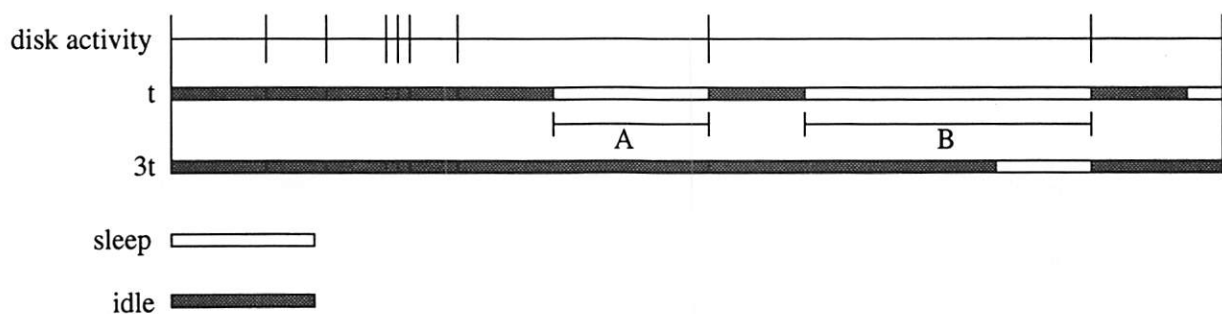


Figure 4: **Effect of Spindown Delay.** This figure illustrates the dual effects of spindown delay on the length of time a disk sleeps.

that the minimum amount of energy is consumed with a spindown delay of around 2 seconds. Using a 2 second spindown delay saves nearly 90% of the 14400 joules that would be consumed if no spindown policy were used. At a spindown delay of zero, there is a sharp increase in energy consumption due to the cost associated with spinning up the disk repeatedly. For our disk drive, the spinup cost is 2 seconds of user delay and 6 joules of energy. The fact that a few second delay is effective confirms the intuition that disk events usually occur in pockets of a few seconds and that it would be a bad idea to always spin down the disk immediately.

The unintuitive result of these figures is the sharp slope to the right of 2 seconds. The steepness of this region indicates that a very small increase in spindown delay beyond a few seconds will yield a very large energy penalty. With the current practice in industry spinning down the disk after 3-5 minutes, our results show that a factor of four in energy savings can be achieved by reducing the spindown delay to 2 seconds. This translates into added latency in accessing the disk, but as we will show in section 4.2, this penalty is small.

It is insightful to compare our 2 second spindown delay policy with the prescient spindown policy (OPTIMAL_DEMAND in [DKM94]) which uses foreknowledge of disk events to optimally decide when to spindown the disk. Using the numbers from Table 2 and 3, we can compute the period of time a disk must remain idle before the cost of spinning up the disk is below that of just idling. The breakeven point (relative to energy) comes out to be 6.2 seconds. If there is no disk activity for greater than 6.2 seconds, then spinning down the disk will save energy. We can approximate the effect of an optimal spindown policy by examining the energy usage and number of spindowns using a fixed 6 second spindown delay. Note that the only difference between optimal and a fixed 6 second delay occurs when the next disk event occurs more than 6 seconds into the future. If the idle time is less than 6 seconds, neither policy will spindown the disk. If the idle time is more than 6 seconds, both will spindown the disk, but optimal will spin down immediately, instead of waiting 6 seconds before spinning down. Thus the energy consumption for the optimal policy is the energy consumption of a fixed 6 second delay policy, minus the energy consumed by the unnecessary idle time before every spindown. As shown in Figure 5, the 6 second delay policy resulted in about 100 spindowns per session. Multiplying the number of spindowns by 6 seconds at 1 watt of power consumption in idle mode yields 600 joules that prescient knowledge saves over our 6 second spindown delay policy. Thus, the prescient algorithm saves only 30% of the energy used by our fixed spindown delay policy. Since the 2 second spindown delay algorithm already saves 90% of the power that would be consumed with no spindown policy, the prescient algorithm can at best save an additional 3%. This shows that attempts to further reduce energy consumption by refining the spindown policy will not yield much benefit.

4.1.2 Spindown Delay Analysis

The shape of the curve in Figure 2 is somewhat baffling at first sight. Why should a small change in the left end of the curve yield such drastic changes in energy consumption while the right end is relatively flat? The answer lies in the fact that spindown delay has two effects on energy consumption. The first

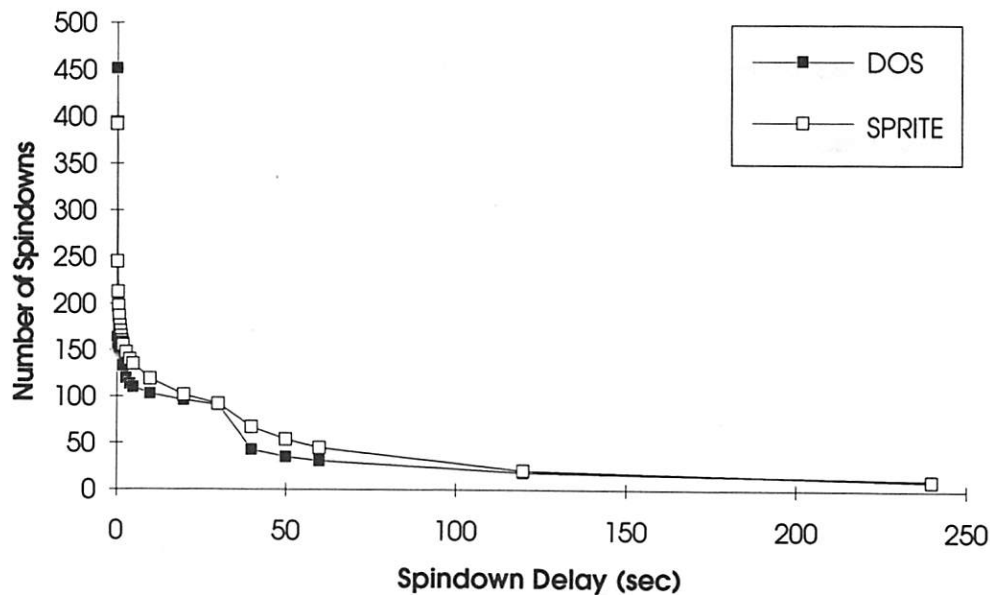


Figure 5: **Number of Spindowns vs. Spindown Delay.** This figure shows the average number of spindowns performed for various spindown delays. The simulations were run with a one megabyte disk cache, 30 second write delay, name-attribute caching enabled and 4 hour traces.

is how often a disk gets to sleep and the second is how long a disk gets to sleep. Figure 4 shows both of these effects. The top line of the figure indicates disk activity and the other two bars illustrate the idle and sleep times of a disk with spindown delay t and $3t$. In the region labeled A, the next disk event arrives sufficiently late that the smaller spindown delay is able to take advantage of the inactivity and spindown the disk. This is not the case with a spindown delay of $3t$. Here, the next disk event is considered part of the disk activity of the preceding cluster and spinning down of the disk occurs $3t$ time units after this last event. The goal of choosing a spindown delay value is to make it small enough that it will define clusters of disk activity, but large enough that the amount of energy saved while sleeping is significantly greater than the cost of spinning up the disk. A zero spindown delay defines a cluster as a single disk event while a fifteen minute spindown delay effectively defines a cluster as the entire session. Our analysis shows that a two second spindown delay effectively determines the extent of a disk cluster.

The second and more important effect of spindown delay on energy consumption is the length of time a disk gets to sleep. The region labeled B shows that the disk sleeps for a much longer period of time with the smaller spindown delay. By reducing the delay, we increase the sleep time, which reduces our energy consumption. This energy savings is then compounded by the number of spindowns per session. As shown in Figure 5, a spindown delay of 2 seconds results in approximately 100 spindowns per session. By increasing the spindown delay from 2 seconds to 2 minutes, one can see how this effect is multiplied to yield the steep slope observed on the left end of figure 2.

4.2 User Delay

Section 4.1 showed that it is possible to reduce the energy consumption of a disk drive to a very small fraction of what it uses without power management. In this section, we analyze the tradeoff between energy consumption and user delay. We define user delay as the sum of the spinup delays over the entire 4 hour trace which are synchronous with the user's activities. These are the delays that the user will feel in his interaction with the computer. Asynchronous spinups due to delayed writes from the disk cache are not counted in user delay, as this activity is transparent to the user.

Figure 6 captures the tradeoff between total user delay over the trace and energy consumption. The figure shows that by tolerating a small amount of user delay, large energy savings can be achieved. The

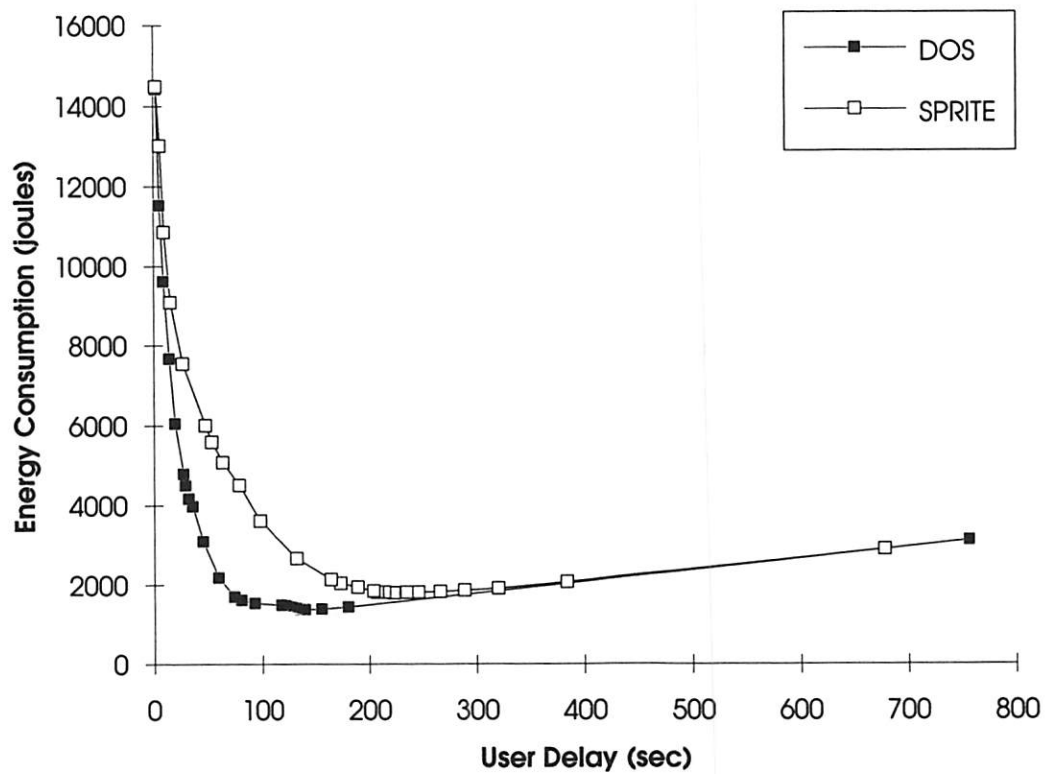


Figure 6: **User Delay vs. Energy Consumption.** This figure shows the tradeoff between user delay and energy consumption. The simulations were run with a one megabyte disk cache, 30 second write delay, name-attribute caching enabled and 4 hour traces. Obtained by varying spindown delay and plotting energy consumption vs. user delay for each data point.

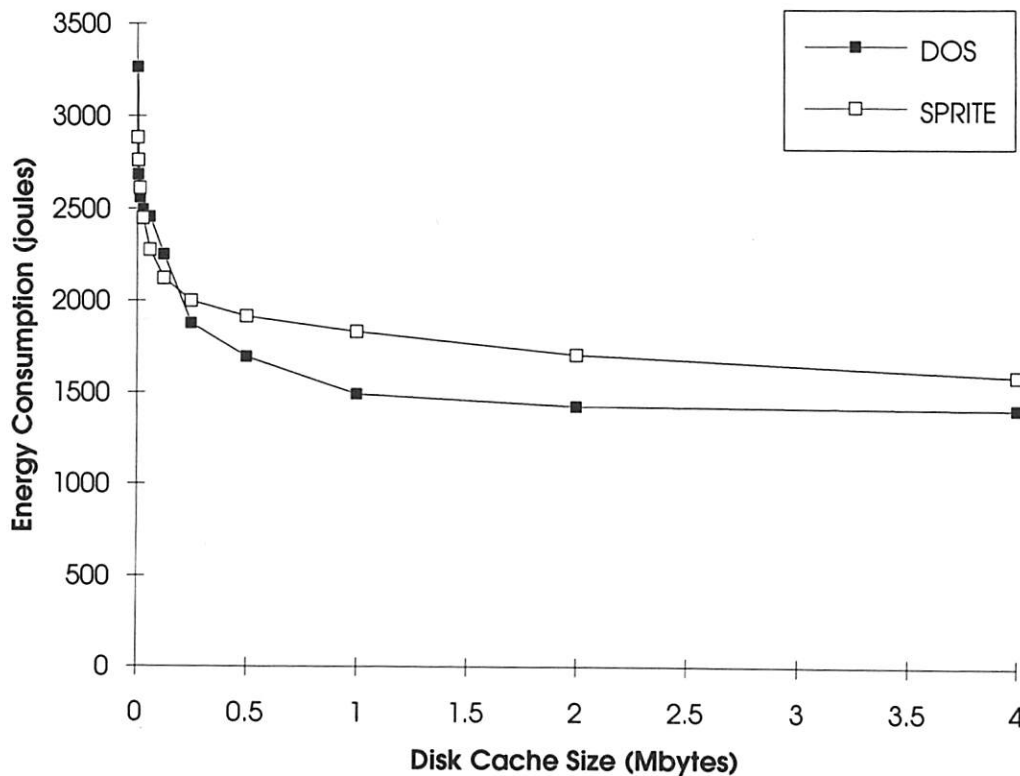


Figure 7: **Disk Cache Size vs. Energy Consumption.** This figure shows the effect of varying disk cache size on energy consumption. The simulations were run with a 2 second spindown delay, 30 second write delay, name-attribute caching enabled and 4 hour traces.

DOS curve bottoms out at around 60 seconds while the Sprite curve bottoms out at around 120 seconds of user delay. These are the delays which would be felt by a user operating with a 2 second spindown delay over a 4 hour period. This amounts to 15-30 seconds of user delay per hour. In other words, the user would have to wait for a 2 second disk spinup 8-15 times per hour. We believe this delay is sufficiently small that its overall effect will be lost in the overhead of the system and application software. Thus a 2 second spindown delay does not impose a significant performance penalty.

4.3 Disk Cache

Contrary to our initial expectations, the disk cache did not exhibit as large an impact on the results as we predicted. Although having a disk cache does aid in reducing energy consumption by filtering the disk traffic, its effect is secondary to the choice of spindown delay.

Figure 7 shows the effects of varying the disk cache size on energy consumption. The figure indicates that a one megabyte cache is sufficient to achieve most of the energy benefits of disk caching. The cost of not having a disk cache is a twofold increase in energy consumption. From a power management point of view, trading the energy consumed by one megabyte of DRAM for the additional energy savings of the disk is a wise investment.

Varying the delay in writing dirty blocks to disk has approximately the same effect on energy consumption as varying the disk cache size. As Figure 8 shows, there is approximately a factor of two decrease in energy consumption by enforcing a write delay of 30 seconds. Again, the curve is very steep at the left end and nearly flat at the right end. This indicates that a very small sacrifice in tolerance to lost data will initially yield large energy savings. However, further sacrifices beyond 60 seconds will not yield any additional savings. Thus, the policy implemented by Unix and Sprite of delaying writes for 30 seconds has applicability in the low power environment as well.

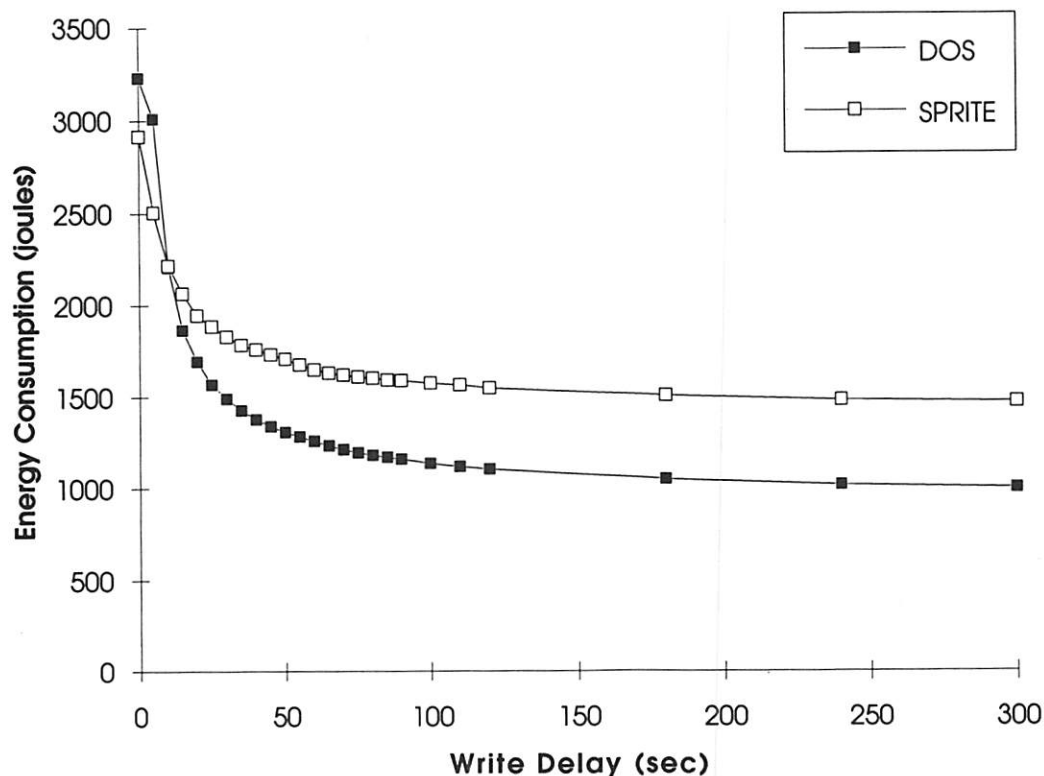


Figure 8: **Write Delay vs. Energy Consumption.** This figure shows the effect of varying write delay on energy consumption. The simulations were run with a 2 second spindown delay, one megabyte disk cache, name-attribute caching enabled and 4 hour traces.

4.4 Spinups

Another area of concern in disk power management is the increased number of disk spinups per session. Increasing the number of spinups per session increases the friction induced wear on the disk-head interface. This in turn decreases the useful lifetime of the disk drive. Figure 5 shows the number of spinups per session under our policies. With a 2 second spindown delay, there are approximately 100 spinups per 4 hour period. Traditional contact start/stop technology disk drives are currently specified at 40,000 start/stops. This gives the moderately disappointing result of 400 four hour sessions before drive wear becomes a problem. However, the portable computer and embedded systems market is currently funding huge efforts into the development of non-contact techniques such as dynamic head loading [PK92] which are targeted specifically for the portable market. These fast, high-shock tolerance drives are expected to populate all future portable computers. The non-contact technology is expected to increase the number of starts/stops of disk drives to one million. With this figure, the expected time to failure under a 2 second spindown delay policy is 10,000 four hour sessions, or about 4 1/2 years of continuous use.

4.5 Name-Attribute Cache

The results of the DOS and Sprite traces turned out to be surprisingly similar from a power management viewpoint. Although Sprite traces consistently consumed more energy and incurred higher user delays than the DOS traces, the shapes of the curves were nearly identical. The only notable point of difference is the relative effectiveness of the name-attribute cache in reducing energy consumption. Table 4 indicates that name-attribute caching is somewhat useful in the DOS environment, but indispensable in the Sprite environment. This difference is due to the presence of large numbers of periodic lookup operations in the Sprite traces. We believe this anomaly is an artifact of the Sprite implementation and is not a general property of a Unix file system. We therefore conclude that a name-attribute cache is moderately successful

Name-Attribute Cache	DOS (joules)	Sprite (joules)
Enabled	1488	1829
Disabled	1800	9149

Table 4: Relative effectiveness of the name-attribute cache under DOS and Sprite. The simulations were run with a 2 second spindown delay, one megabyte disk cache, 30 second write delay and 4 hour traces.

in reducing power consumption, but that its primary contribution to the system is improved performance.

5 Conclusion

We have performed a quantitative analysis of the costs and benefits of spinning down a disk drive as a power management technique. Given an intelligently designed operating system with a one megabyte write-back disk cache and a twenty directory name-attribute cache, we have shown that 90% of the energy consumed by a disk drive can be eliminated with almost no performance or reliability impact. Furthermore, we determined that a spindown delay of 2 seconds minimizes the power consumption of the disk and that deviation from this minimum rapidly increases the energy consumption without significant gains in performance.

6 Acknowledgements

This work was supported in part by the National Science Foundation (CDA-8722788), the Digital Equipment Corporation (the Systems Research Center and the External Research Program), and the AT&T Foundation. Anderson was also supported by a National Science Foundation Young Investigator Award.

We would like to thank Albert Goto who diligently collected and labeled the hundreds of DOS traces taken from the CCSF computing laboratory. This study would have suffered greatly without his efforts.

References

- [BADOS92] M. Baker, S. Asami, E. Deprit, J. Ousterhout and M. Seltzer, "Non-volatile memory for fast, reliable file systems" *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1992, 10-22
- [BHK91] M. Baker, J. Hartman, M. Kupfer, K. Shirriff and J. Ousterhout, "Measurements of a Distributed File System" *Proceedings of the 13th Symposium on Operating System Principles*, Oct. 1991, 198-212
- [Compaq] Compaq User's Manual, 1993
- [CDLM93] R. Cáceres, F. Douglass, K. Li and B. Marsh, "Operating Systems Implications of Solid-State Mobile Computers" *Proceedings of the Fourth Workshop on Workstation Operating Systems*, Oct. 1993, 21-27
- [DM93] F. Douglass and B. Marsh, "Low-power Disk Management for Mobile Computers", Technical Report MITL-TR-53-93, Matsushita Information Technology Laboratory, April 1993
- [DKM94] F. Douglass, P. Krishnan and B. Marsh, "Thwarting the Power-Hungry Disk" *USENIX*, Winter 1994

- [Greenawalt94] P. Greenawalt, "Modeling Power Management for Hard Disks", to appear in *Proceedings of the 13th Symposium on Modeling and Simulation of Computer and Telecommunication Systems* 1994
- [LPCMOS] Low Power CMOS Digital Design, IEEE Solid State, April 1992
- [Maxtor] Maxtor Corporation, "The MXL-105-III" 1993
- [MDK94] B. Marsh, F. Douglass and P. Krishnan, "Flash Memory File Caching for Mobile Computers", to appear in *Proceedings of the 27th Hawaii Conference on Systems Sciences IEEE* 1994
- [PK92] T. Parrish and G. Kelsic, "Dynamic Head Loading Technology Increases Drive Ruggedness" *Computer Technology Review*, Winter 1992
- [RW93] C. Ruemmler and J. Wilkes, "UNIX disk access patterns" *Proceedings of the Winter 1993 USENIX Conference* Jan. 1993, 405-420
- [SO92] K. Shirriff and J. Ousterhout, "A Trace-Driven Analysis of Name and Attribute Caching in a Distributed System", *USENIX*, Winter 1992
- [Wilkes92] J. Wilkes, "Predictive power conservation", Technical Report HPL-CSP-92-5, Hewlett-Packard Laboratories Feb. 1992

Author Information

Kester Li is a graduate student in the Department of Electrical Engineering and Computer Sciences at the University of California at Berkeley. His interests include programming languages and operating systems. He received a B.A. in computer science from UC Berkeley in 1992. Kester Li can be reached at "kesterl@cs.berkeley.edu".

Roger Kumpf is a graduate student in the Department of Electrical Engineering and Computer Sciences at the University of California at Berkeley. His interests include software systems and user interfaces. He received a B.S. in computer science from The Ohio State University in 1992. Roger can be reached by e-mail at "kumpf@cs.berkeley.edu".

Paul Horton is a graduate student in the Department of Electrical Engineering and Computer Sciences at the University of California at Berkeley. He is interested in computation biology and artificial and natural intelligence. He received his B.S. in molecular biology from The University of Washington in 1989 and his M.S. in biophysics from Kyōto University in 1992. He can be reached at "paulh@cs.berkeley.edu".

Thomas Anderson is an Assistant Professor in the Computer Science Division at the University of California at Berkeley. He received his A.B. in philosophy from Harvard University in 1983 and his M.S. and Ph.D. in computer science from the University of Washington in 1989 and 1991, respectively. He won an NSF Young Investigator Award in 1992, and he co-authored award papers at the 1989 SIGMETRICS Conference, the 1989 and 1991 Symposia on Operating Systems Principles, the 1992 ASPLOS Conference, and the 1993 Winter and Summer USENIX Conferences. His interests include operating systems, computer architecture, multiprocessors, high speed networks, massive storage systems, and computer science education. His e-mail address is "tea@cs.berkeley.edu".

Thwarting the Power-Hungry Disk

(Errata)

*Fred Douglass
P. Krishnan
Brian Marsh*

We recently discovered an error in the simulations reported in the paper entitled "Thwarting the Power-Hungry Disk." For the Macintosh Powerbook trace, writes into the DRAM buffer cache were not properly propagated to the disk. As a result, the total power consumed by the disk subsystem was reported to be lower than it should have been. The frequency of read spin-up events was higher than it should have been, since some reads that the original simulations indicated would cause a spin-up would instead occur with the disk already spinning. The benefit of the optimal algorithm was somewhat understated because there were fewer disk events than should have occurred.

The overall conclusions of the paper are unaffected, and many of the changes are relatively minor. However, a couple of reported numbers changed significantly; also, in the process of correcting these numbers we found one measurement that was reported erroneously. In addition to shifts within Figure 1, which is not reproduced here, a list of the affected parts of the paper follows:

Abstract The optimal algorithm reduces power consumption from 35–55%, not 35–50% as reported. (This change also applies in Sections 5, 6, and 7.) The 10-second threshold for the Powerbook trace and Go•Drive disk reduces energy consumption by 31%, not 40%. It results in 120 additional delays, not 140.

Section 5 The Powerbook trace on the Kittyhawk shows an 8% improvement in energy with the 1-second threshold, compared to the 5-second threshold, with a 140% increase in delays.

Section 5.1 For the Powerbook trace, OPTIMAL_OPTIMAL used 47% and 45% of the energy of the recommended thresholds for the Kittyhawk and Go•Drive, respectively. This compared to 62% and 46% in the original paper.

Section 5.2 The 11% power reduction mentioned in this section is erroneous. The 10-second spin-down threshold for the Powerbook trace on the Go•Drive saved 39% of the power consumed by the 5-minute spindown policy in the original simulation, and 31% in the corrected simulation. It consumes 50% more power than the optimal algorithm. With the Powerbook trace on the Kittyhawk, moving from 1s to 5s increases energy consumption by 8% but reduces read spin-up delays by 41%.

Section 6 The absolute count of read spin-up delays changed slightly, from 388 to 396. (Also, we have learned that our description of Li, *et al.* [12] was not an appropriate comparison, as they report that 90% of disk power consumption can be eliminated *compared to never spinning down the disk.*)

Section 7 Threshold policies that spin down after 1–10 seconds come within 7–52% of the off-line policy, and consume 56–92% of that consumed by the recommended thresholds.

We apologize for the errors and thank the USENIX Association for the opportunity to provide corrections.

Thwarting the Power-Hungry Disk

Fred Douglass

Matsushita Information Technology Laboratory

P. Krishnan

Brown University

Brian Marsh

Matsushita Information Technology Laboratory

Abstract

Minimizing power consumption is important for mobile computers, and disks consume a significant portion of system-wide power. There is a large difference in power consumption between a disk that is spinning and one that is not, so systems try to keep the disk spinning only when it must. The system must trade off between the power that can be saved by spinning the disk down quickly after each access and the impact on response time from spinning it up again too often. We use trace-driven simulation to examine these trade-offs, and compare a number of different algorithms for controlling disk spin-down. We simulate disk accesses from a mobile computer (a Macintosh Powerbook Duo 230) and also from a desktop workstation (a Hewlett-Packard 9000/845 personal workstation running HP-UX), running on two disks used on mobile computers, the Hewlett-Packard Kittyhawk C3014A and the Quantum Go•Drive 120. We show that the “perfect” off-line algorithm—one that consumes minimum power without increasing response time relative to a disk that never spins down—can reduce disk power consumption by 35–50%, compared to the fixed threshold suggested by manufacturers. An on-line algorithm with a threshold of 10 seconds, running on the Powerbook trace and Go•Drive disk, reduces energy consumption by about 40% compared to the the 5-minute threshold recommended by manufacturers of comparable disks; however, over a 4-hour trace period it results in 140 additional delays due to disk spin-ups.

1 Introduction

The recent trend toward portable, battery-operated computers is motivating advancements in reducing power consumption through both hardware and software approaches. One area that has seen rapid improvement is disks for mobile computers: decreasing scale and increasing density have led to small, lightweight, low-power disks such as the Hewlett-Packard Kittyhawk [9], as well as flash memory devices such as Seagate’s IDE-compatible FlashDrive [16]. In recent papers we have examined the effect of using some amount of flash memory as a cache of frequently-accessed disk blocks, in order to keep the disk from spinning as often [13], or as a complete replacement for disk [3]. Until flash is inexpensive¹ and long-lasting enough to be ubiquitous, many notebook and laptop computers will have only DRAM and a hard disk. If the disk drive is used with any frequency, it will have a significant impact on the length of time the computer can operate on a single battery charge.

Spinning down the disk when it is not being used can save power. Most if not all current mobile computers use a fixed threshold to determine when to spin down the disk: if the disk has been idle for some (predetermined) amount of time, the disk is spun down. The disk is spun up again upon the next access. The fixed threshold is typically on the order of many seconds or minutes to minimize the delay

¹ A recent advertisement in the New York Times priced the Hewlett-Packard Omnibook at \$1599 with a 40-Mbyte hard disk or \$1799 with a 10-Mbyte flash memory card. These prices have dropped a few hundred dollars just since the introduction of the Omnibook, with the gap between them closing, but even so, the added cost of flash is substantial when the relative sizes of the media are considered.

Machine	CPU Speed (MHz)	Disk Size (MBytes)	Disk State	System Power (W)	Power Savings (W)	% of Total System Power
Zenith Mastersport SLe	25.0	85	Idle	10.5	1.0	9.5
	6.5		Stopped	9.5		
Toshiba T3300SL	25.0	120	Idle	8.1	1.2	14.8
	6.5		Stopped	6.9		
Dell 320SLi	20.0	120	Idle	4.5	0.9	20.0
	2.5		Stopped	3.6		
			Idle	3.2	1.0	31.2
			Stopped	2.2		

Table 1: Power measurements of three typical laptop computers.

from on-demand disk spin-ups. The Hewlett-Packard Kittyhawk C3014A spins down and up again in about three seconds, and its manufacturer recommends spinning it down after about five seconds of inactivity [10]; most other disks take several seconds for spin-down/spin-up and are recommended to spin down only after a period of minutes [5, 18]. In fact, spinning a disk for just a few seconds without accessing it can consume more power than spinning it up again upon the next access. Spinning down the disk more aggressively may therefore reduce the power consumption of the disk, in exchange for higher latency upon the first access after the disk has been spun down.

To understand this tradeoff, we use trace-driven simulation to evaluate different disk spin-down policies for reducing power consumption. We consider threshold policies, which are practical to implement, off-line algorithms that are optimal in terms of power consumption, and predictive on-line algorithms that take past history into account. We find that threshold policies that spin down the disk after 1–10 seconds come close to the power consumption of the optimal off-line algorithm, which reduces the power consumption using manufacturers' recommended thresholds by about half. However, in some cases the threshold algorithms substantially increase the delays incurred by the user. These delays could be avoided if access times could be predicted accurately enough, but predictive strategies that close the gap between the optimal off-line algorithm and current threshold-based algorithms are difficult to construct.

The rest of this paper is organized as follows. Section 2 elaborates on the motivation behind our work, specifically the power consumed by the disk subsystem in current mobile computers. Section 3 discusses various spin-down policies. Section 4 describes the input traces and simulator used in our experiments, and Section 5 reports the results of our simulations. Section 6 discusses related work, and finally, Section 7 concludes the paper.

2 Power Consumption

To get some idea of how the disk can affect battery life, we measured the power consumption of the disk on a Dell 320 SLi, a Toshiba T3300SL, and a Zenith Mastersport SLe. This data is shown in Table 1.² All three machines are running Mach 3.0 (UX37/MK77). The machines are listed in the relative order of their age. All were purchased in the past two years, and at the time represented the state-of-the-art in low-power notebook design. All three use the Intel SL Superset, which consists of the 386 SL CPU and the 82360 I/O controller. The Zenith and the Toshiba both have a backlit LCD display, while the Dell uses a "triple super-twist nematic, reflective LCD" display.

The measurements were made using an HP 34401A multimeter using customized instrumentation

²This table also appears in [13].

software. We varied two parameters: the speed of the CPU and the state of the disk. We controlled the state of both using hot-key bindings supplied by the system manufacturers. The CPU speed was set at the fastest and slowest speeds available. The disk was set to be either "spun-up" or "spun-down."

Varying the clock speed is important because the CPU can consume a large amount of power. Reducing its clock speed when there is no work to be done can significantly reduce the amount of power consumed. In fact, many commercial systems already incorporate "Advanced Power Management" [6] to take advantage of such savings. As a result, it is reasonable to expect that on most systems the CPU will be slowed down when idle. Mobile computers are likely to be used for highly interactive software (such as mailers, news readers, editors, etc.) so it is reasonable to expect a large amount of CPU idle time. When the CPU clock speed is reduced, a spinning disk will consume proportionally more of the total system power.

There are several important things to note about Table 1. First, disk densities are increasing, making it possible to carry more data. Machines are now available with even larger disks than the systems we instrumented. Second, even though disk densities have increased, the power used by the largest disks has stayed about the same, around 1W for an idle spinning disk. Third, the overall system power cost is dropping. The result is that the amount of power consumed by the disk sub-system on these notebook computers has increased from 9% to 31%. Improved recording densities make it possible to store more data on the same physical device, but they do not affect the physical mass. Drives are becoming more efficient, but cost about the same to spin up and to keep spinning. Theoretically, machines could have smaller disks, but in practice, higher recording densities are used to increase the overall capacity of the storage system instead of decreasing its power consumption. With the exception of the smallest and lightest computers, such as the Hewlett-Packard Omnibook [11], the trend seems to be to carry a larger disk with the same mass rather than a smaller disk with the same number of bytes.

Our measurements suggest that proper disk management can improve battery life. In addition, technology trends suggest that such improvements will become increasingly important. For instance, battery life for the Dell 320 could be improved 20 to 31%, the amount that could be saved if the disk were off all the time. Put another way, a battery that lasts 5 hours could last from 6 to 6.5 hours instead. Of course, turning the disk off can result in increased access latency, so policies for saving power need to balance the two issues. In the next section, we describe different approaches to managing this tradeoff.

3 Policies

We investigated two types of algorithms for spinning a disk up and down: *off-line*, which can use future knowledge, and *on-line*, which can use only past behavior. Off-line algorithms are useful as a baseline for comparing different on-line algorithms, and for showing where there is room for potential improvement. On-line algorithms are implementable (though some may consume more memory, processing, or other resources than is feasible). Typically mobile computers spin down their disk based on a simple heuristic; for instance, when it has not been accessed in a predetermined period of time (such as 5 minutes). They spin up the disk when the first access after a spin-down occurs.

3.1 Off-line Policies

Spin-up and spin-down policies should minimize both power consumption and response time. Unfortunately, power and time are not always optimized by the same policy. It is easy to see that the optimal policy with respect to response time is not necessarily optimal with respect to power consumption. Leaving the disk spinning all the time will produce the minimal impact on response time, but will waste power if the disk isn't accessed for long periods of time. Likewise, the optimal policy with respect to power may result in a delay when a new request stalls waiting for the disk to spin up.

An off-line policy for spinning down the disk is based on the relative costs of spinning or starting it up. We define T_d as the amount of time the disk must spin before the cost of spinning the disk continuously equals the cost of spinning it down immediately and then spinning it up again just prior to the next access. With future knowledge one can spin down the disk immediately if the next access will take place more than T_d seconds in the future. This will result in the minimal power consumption of all spin-down algorithms,

Characteristic	Hewlett-Packard Kittyhawk C3014A	Quantum GoDrive 120
Capacity (Mbytes)	40	120
Power consumed, active, (W)	1.5	1.7
Power consumed, idle, (W)	0.6	1.0
Power consumed, spin up (W)	2.2	5.5
Normal time to spin up (s)	1.1	2.5
Normal time to spin down (s)	0.5	6.0
Avg time to read 1 Kbyte (ms)	22.5	26.7
Break-even interarrival time T_d (s)	5.0	14.9

Table 2: Disk characteristics of the Kittyhawk C3014A and Quantum GoDrive 120. The Kittyhawk has less capacity than the GoDrive, but it has significantly lower operating costs, especially the power drawn during disk spin-up and the average spin-up duration. As a result, the break-even point for the Kittyhawk is about a fourth that of the GoDrive, making a short spin-down threshold much more important for the Kittyhawk. Also, the Kittyhawk spins down in a half a second, while the GoDrive takes "< 6s" [14].

among policies that have minimum response time. There are, of course, complications beyond this simple threshold; for instance, a disk usually has multiple states that consume decreasing amounts of power but from which it is increasingly costly (in time and power) to return to the active state. Table 2 lists the characteristics of two disk drives for mobile computers, the Hewlett-Packard Kittyhawk C3014A [9] and the Quantum GoDrive 120 [14], including values for T_d .

In fact, the time to spin up the disk once a new request arrives has a substantial impact on response time. An on-line algorithm that spins up the disk when a request arrives if the disk is spun down will cause the request to wait until the disk is ready, typically at least 1–2 seconds. This latency is up to a couple of orders of magnitude greater than normal disk access times, and should be avoided whenever possible. The high spin-up overhead is the reason why typical thresholds for spinning down a hard disk are often on the order of several minutes even if T_d is just a few seconds: if the disk has not been accessed for several minutes then the overhead of a couple of extra seconds before a new request can be serviced is neither unexpected nor unreasonable. In contrast to the on-line approach, an off-line algorithm can not only spin down the disk when that would save power, it can spin up the disk again just in time for the next request to arrive.

3.2 Threshold-based Policies

Threshold-based policies are the standard timeout-based algorithms used in most present systems. If the disk is not accessed within a fixed period of time it is spun down. That timeout value may vary depending on environmental characteristics (e.g., running off battery rather than A/C power) or input from the user, but is normally not modified dynamically by the system. The disk is spun up again upon the next access, and the request must wait for spin-up to complete.

3.3 Predictive Policies

The predictive policies store historical information and interpret it to predict the next access. There are many possible heuristics for interpreting this data. For instance, Wilkes hypothesized that it would be effective to use a weighted average of a few previous interarrival times to decide when to spin down the disk on a mobile computer. He noted as well that if inactive intervals were of roughly fixed duration, the disk could be spun up in advance of the expected time of the next operation [17]. If access patterns are not so consistent, however, these techniques may not prove to be helpful.

In practice, predictive models may have difficulty beating simple threshold policies because access patterns are not sufficiently regular. If *every* access either came within 100ms of the previous access or after a delay of many seconds, it would be possible to spin down the disk after 100ms passed. But if even a small fraction of accesses take place between 100ms and T_d seconds after their predecessors, having such a quick

trigger to spin down the disk could be costly. We are presently evaluating some predictive heuristics to see if they can be applied across a range of workloads, and have some comments on predictive algorithms in Section 5.3.

3.4 Taxonomy

Here we describe a taxonomy of disk spin-down policies, considering both the algorithm used to decide when to spin down the disk and the one used to spin it up again. The naming scheme indicates the most salient feature of the particular algorithm; the first part of the name denotes the spin-down policy, while the second part denotes the spin-up policy.

OPTIMAL_OPTIMAL This is the off-line algorithm described in Section 3.1, which uses future knowledge to spin down the disk and to spin it up again *prior* to the next access. It provides the lowest power consumption among policies that have minimum response time. Other off-line algorithms with slightly lower power consumption may exist, but they will suffer increased response times.

OPTIMAL_DEMAND An alternative off-line approach is to assume future knowledge of access times when deciding whether to spin down the disk but to delay the first request upon spin-up. This algorithm will consume about the same amount of power as **OPTIMAL_OPTIMAL**, but will have poorer response time. This algorithm is relevant because an on-line algorithm may be better at predicting that the next request will occur more than T_d seconds in the future than predicting exactly when the request will occur; i.e., predicting the correct time to spin down the disk may be easier than predicting when to spin it up again.

THRESHOLD_DEMAND The disk is spun down after a fixed period of inactivity and is spun up upon the next access. This is the policy used on most systems at present.

THRESHOLD_OPTIMAL The disk is spun down after a fixed period of inactivity but is spun up just before the next access. If the next access occurs too soon (there is not enough time to spin up the disk before the access) then the access will be delayed. This algorithm is primarily for purposes of comparison and completeness.

PREDICTIVE_DEMAND Spin-down is based on a heuristic that uses information about previous accesses to determine when to spin down the disk. Spin-up is performed upon the next access.

PREDICTIVE_PREDICTIVE Spin-down uses the same heuristic as **PREDICTIVE_DEMAND**, and spin-up is based on a predictive heuristic as well.

4 Methodology

4.1 Traces

To evaluate the effect of the disk spin-down policy on power consumption and response time, we used traces from two execution environments: an Apple Macintosh Powerbook Duo 230 and a Hewlett-Packard 9000/845 personal workstation running HP-UX.

The Powerbook traces were gathered at MITL. We collected two traces of approximately two hours of activity and one trace of approximately four hours of activity. One of the two-hour traces has characteristics very similar to the four-hour trace, while the other shows more constant use of the disk. In this paper we report results of simulating the four-hour trace, during which time mostly Microsoft Word, an editor, and Eudora, an electronic mail application, were running.

Disk management on the Macintosh is unusual in several respects. First, its cache behavior is dependent on the size of the cache: a larger disk cache not only increases the number of blocks that can be cached but also increases the maximum size of any given read or write that can be cached. Even with a cache size of 256 Kbytes, the maximum transfer that can be cached is only 8176 bytes [1]. Second, writes that are cached in the buffer cache are later passed to the disk in 512-byte units, which can degrade performance

	Powerbook	HP-UX
Duration	4.1 hours	7 days
Mean interarrival time (s)	0.4	15.6
Standard deviation of interarrival time (s)	1.9	142.9
Maximum interarrival time (s)	269.2	1769.5

Table 3: Summary of trace characteristics. An important distinction between the Powerbook and HP-UX traces is that the statistics for the Powerbook trace report the interarrival times as seen by the buffer cache while the ones for the HP-UX trace are as seen by the disk.

relative to transferring larger files as a unit. Third, a Macintosh may be configured with a “RAM disk” that behaves like a magnetic disk drive but is stored in DRAM. On the Powerbook Duo the RAM disk is not persistent, so it is useful only for storage of temporary files, other noncritical files that can be copied to disk later, or copies of read-only files such as the System folder. The RAM disk thus allows users to get around the deficiencies of the buffer cache, but only for a specific subset of files.

Our Powerbook trace records reflected access at the file-system level (i.e., above the disk cache). Rather than simulating the Macintosh buffer cache as it is implemented, we simulated a simple LRU write-through buffer cache with each 1-Kbyte block handled separately and no maximum per-file limit. The Macintosh enforces a minimum cache size of 32 Kbytes; we varied the cache size from having no cache at all to a maximum of 1 Mbyte. Because a relatively large cache is essential to eliminating enough disk accesses to make spinning down the disk worthwhile [12], in this paper we report results for the 1-Mbyte cache. Also, in our Powerbook traces, about 2% of accesses went to files on the RAM disk, and we ignored these accesses in the simulator. More experienced Powerbook users might have different access patterns, with more accesses to a RAM disk, but in fact one may consider the 1-Mbyte disk cache to be equivalent to a RAM disk of the same size.

In addition, we used traces from an HP-UX workstation, documented by Ruemmler and Wilkes in a previous USENIX conference [15]. We used the HP-UX traces for three reasons: first, because we did not have the Powerbook traces available initially, and even now have somewhat limited experience with those traces; second, because there are a number of UNIX-based mobile platforms available, so a UNIX trace might be indicative of actual mobile usage patterns; and third, because UNIX does the sort of aggressive caching that is essential for eliminating disk accesses. The HP-UX traces are at the disk level; they represent requests from the HP-UX buffer cache to the disk. As a result, we did not simulate a buffer cache for these traces.

We believe the HP-UX traces should be representative of a mobile environment because the sorts of activities mobile users perform—such as word processing, electronic mail, and spreadsheets—are the same activities as the user in the HP-UX trace would perform in the office. Nevertheless, the HP-UX and Powerbook traces do vary considerably in some respects. Most notably, the HP-UX trace is over a prolonged period of time, with about the same number of accesses as the Powerbook trace had over four hours spread out instead over a week’s time. (The original trace consisted of two months’ worth of data, and we used the first week’s worth.) The HP-UX trace has pauses of up to a half-hour between accesses, so any policy that spins the disk down at all will result in spin-ups after such long pauses. However, there are still periods of activity within that trace that result in large differences between different spin-down policies. Table 3 summarizes some characteristics of the two traces.

4.2 Simulator

The simulator consists of roughly 7500 lines of C code (including comments). It is a general storage management simulator that models three levels of a storage hierarchy, nominally DRAM, flash memory, and magnetic disk; for these experiments the size of flash was always set to 0. For the HP-UX trace, the size of the DRAM cache was also set to 0, as described above, so all disk requests from the original trace resulted in disk requests in the simulations.

Each disk is represented by a file specifying a number of parameters, one for each state in which the disk might be (reading, writing, active, idling, spun down/standby, or halted), and one for each state transition. The configuration files specify how long the disk stays in a given state or transition and how much power, in Watts, is consumed during that time. Several of these parameters are shown in Table 2.

We made a number of simplifying assumptions in the simulator. First, a disk access is assumed to take the average time for seek and rotational latency, unless it involves a disk block with a numerical identifier within ϵ of the previous block accessed. Second, all operations and state transitions are assumed to take the average or "typical" time specified by the manufacturer, if one is specified, or else the maximum time. While these assumptions may affect the specific values of energy and response time produced by the simulator, we do not believe they affect the relative differences in energy consumption and response time from using different spin-down policies.

5 Results

We simulated a number of threshold-based policies as well as the `OPTIMAL_OPTIMAL` and `OPTIMAL_DEMAND` policies, running on both the Powerbook and HP-UX traces, and using both the Kittyhawk and Go•Drive specifications. (We have also experimented with predictive algorithms; some preliminary results are discussed below in Section 5.3.) In this paper we consider two metrics:

Energy Consumption The number of Joules consumed by the disk over the course of the simulation.

Read-Spin-up Delays The number of times a read operation was delayed to spin the disk up.

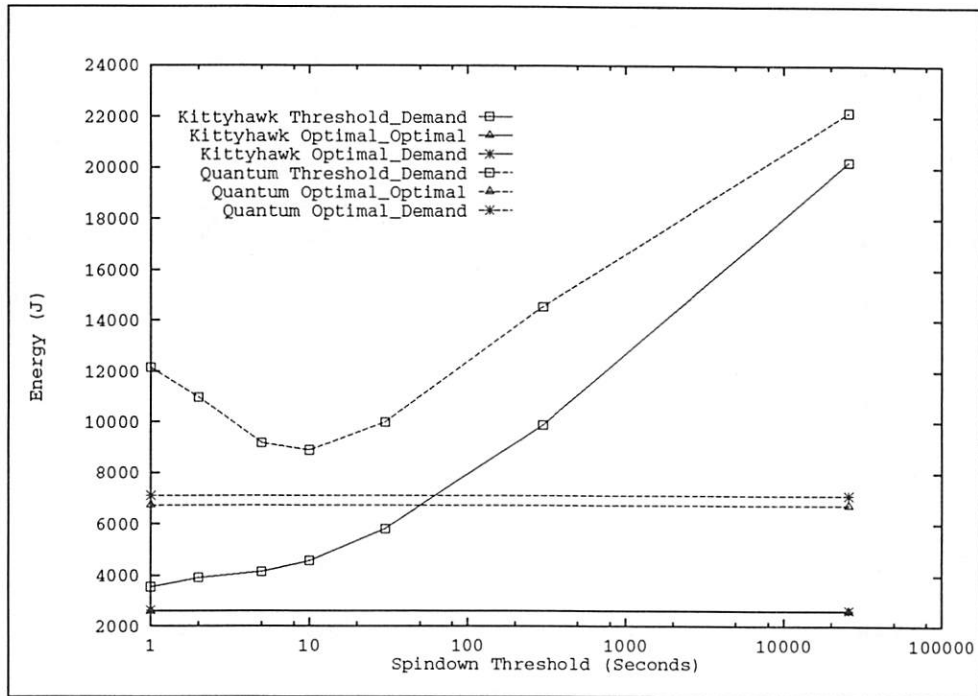
Another metric for the impact on read response time might be the average response time across all reads, but that metric is less satisfying: it considers average delay but not the great discrepancy between operations that have no spin-up delay and those that are delayed. In fact, the actual delay from spin-up varies from about 1 second on the Kittyhawk to 2.5 seconds on the Go•Drive, so the penalty from these undesirable spin-up delays is much greater for the Go•Drive.

Note also that we do not report the impact on write operations, since writes are usually asynchronous, and because even synchronous writes can be decoupled from disk latency with a small amount of nonvolatile memory [2, 15].

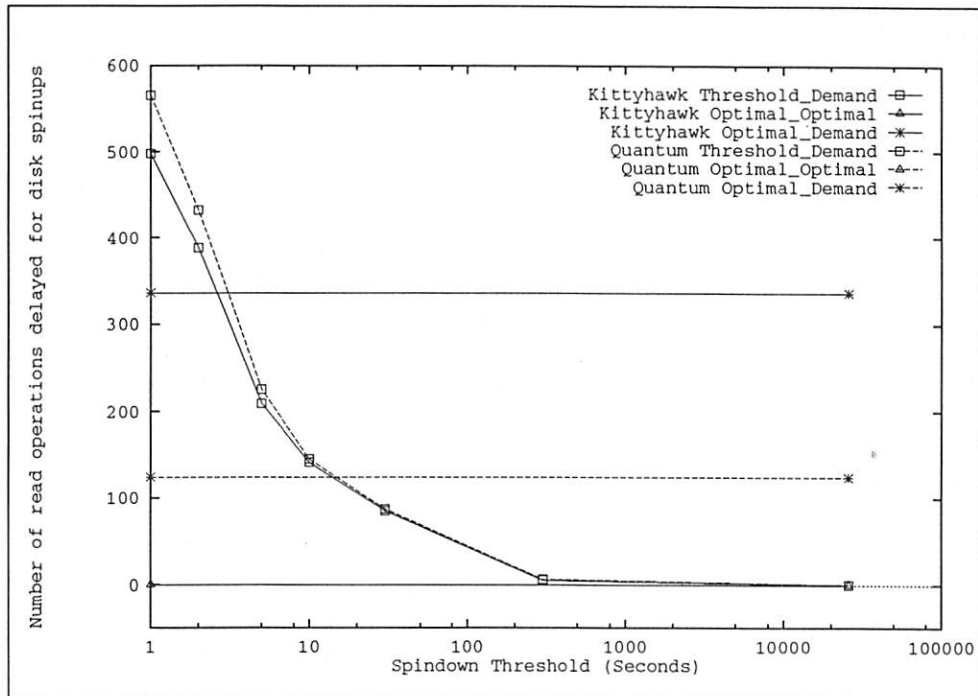
Figure 1 shows the energy consumption and read-spin-up delays for the Powerbook traces, with a 1-Mbyte cache, and Figure 2 shows the same for the HP-UX traces (which has an implicit buffer cache, as discussed above). Both figures show, for both types of disk, several `THRESHOLD_DEMAND` policies, `OPTIMAL_OPTIMAL`, and `OPTIMAL_DEMAND`. For the threshold-based policies, the disk always goes to the idle state after two seconds if it has not already spun down by then, and always goes from the "standby" (spun-down) state to the "halt" state immediately. On the Kittyhawk there is marginal overhead in going from "halt" to "active" relative to going from "standby" to "active," and on the Go•Drive the "standby" and "halt" states are identical.

The most important conclusions one may reach from these figures are:

- The off-line `OPTIMAL_OPTIMAL` algorithm can reduce disk power consumption by 35–50%, compared to the fixed threshold suggested by manufacturers, without adversely affecting response time. (This compares `OPTIMAL_OPTIMAL` to the 5-second spindown of the Kittyhawk and the 5-minute spindown of the Go•Drive 120.)
- On-line `THRESHOLD_DEMAND` algorithms with shorter than recommended thresholds approach the power consumption of `OPTIMAL_OPTIMAL` but may increase the number of read-spin-up delays substantially.
- The best compromise between power consumption and response time is workload-dependent. For the HP-UX trace on the Kittyhawk, a spin-down threshold of 1s consumes 23% less power than the

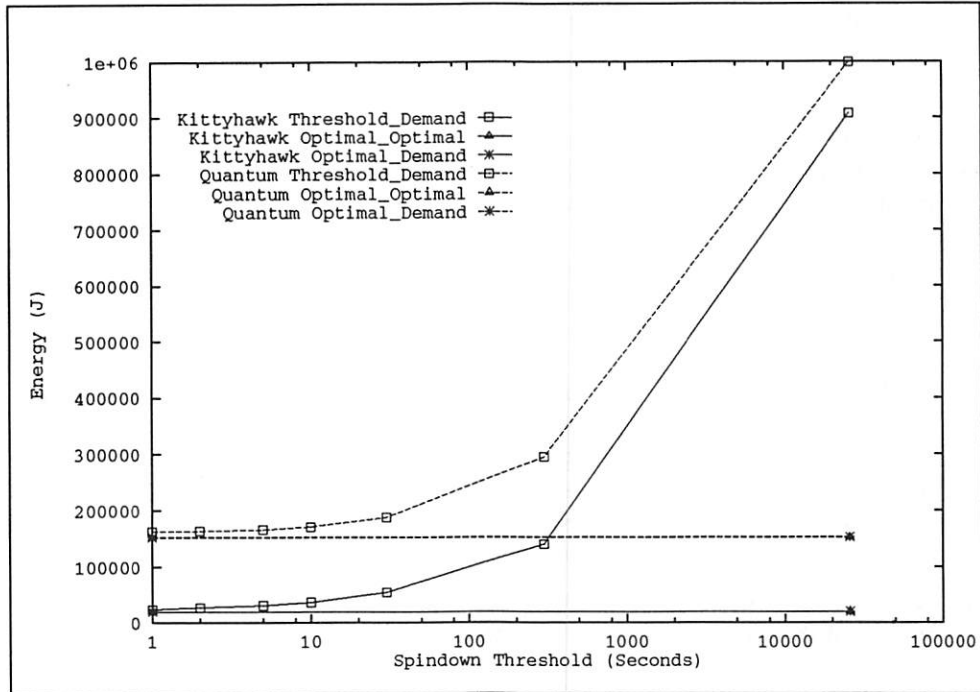


(a) Energy consumption as a function of spin-down time.

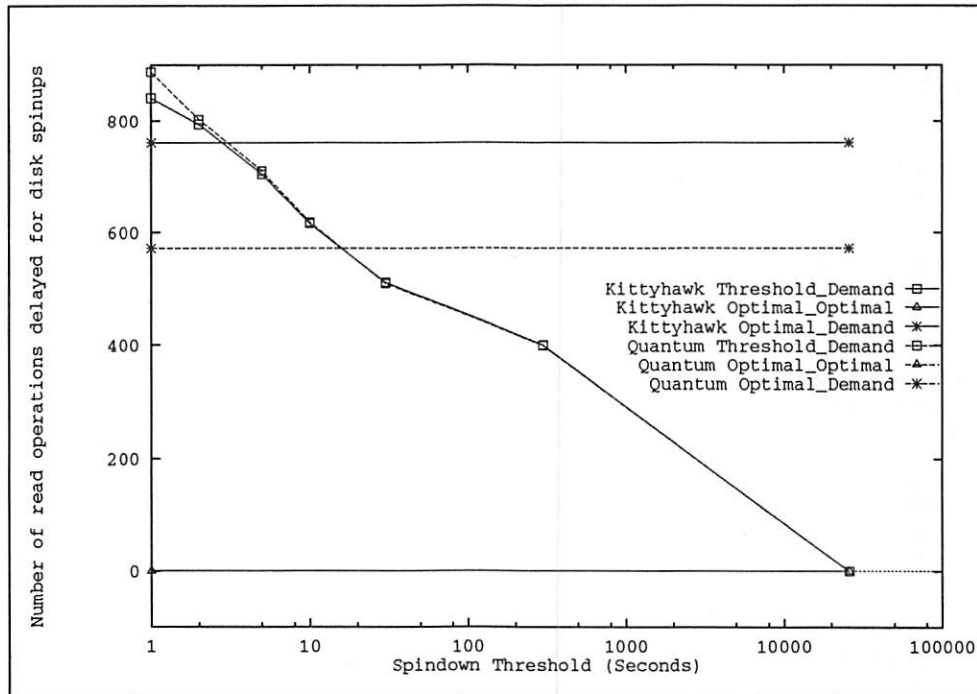


(b) Delays due to spin-up on read accesses, as a function of spin-down time.

Figure 1: Results of simulating the 4-hour Powerbook trace on Kittyhawk and GoDrive 120 disks.



(a) Energy consumption as a function of spin-down time.



(b) Delays due to spin-up on read accesses, as a function of spin-down time.

Figure 2: Results of simulating the HP-UX trace on Kittyhawk and Go•Drive 120 disks.

recommended threshold of 5s, and increases delays by 19%, a fair tradeoff. The Powerbook trace on the same hardware shows a 15% improvement in energy with the 1-second threshold but a 150% increase in delays.

- Lastly, the characteristics of the disk make an enormous difference in the appropriateness of an aggressive spin-down policy. The high latency to spin down and spin up the Go•Drive 120, compared to the Kittyhawk, results in a higher value for T_d and, for the Powerbook trace, minimal power consumption for a threshold of 10s rather than 1s for the Kittyhawk.

We discuss each type of algorithm in turn, as well as the impact of disk characteristics.

5.1 Off-line Algorithms

With future knowledge of disk activity, one can both reduce energy consumption and delays due to disk spin-up. `OPTIMAL_OPTIMAL` uses 64% of the energy consumed by the 5-second `THRESHOLD_DEMAND` policy for the HP-UX trace running on the Kittyhawk; it uses 52% of the energy consumed by the 5-minute policy running on the Go•Drive. For the Powerbook trace, `OPTIMAL_OPTIMAL` used 62% and 46% of the energy of the recommended thresholds for the Kittyhawk and Go•Drive, respectively. In each case, because the disk was always spinning at the time of the next request, response time would improve as well.

`OPTIMAL_DEMAND` considers the hypothetical case where one could predict the future well enough to spin down immediately if that would save power, but would not be able to predict the time of the next access precisely. It uses about the same amount of energy as `OPTIMAL_OPTIMAL` but has a much larger number of read-spin-up delays than the 5-minute `THRESHOLD_DEMAND` policy, since many more read operations result in the disk spinning up. The number of read-spin-up delays incurred by this algorithm is a lower bound on the number of delays that a `THRESHOLD_DEMAND` algorithm with a spin-down threshold less than T_d would incur, and an upper bound on the number that a `THRESHOLD_DEMAND` algorithm with a spin-down threshold greater than T_d would incur.

5.2 Threshold-Demand Algorithms

The set of `THRESHOLD_DEMAND` algorithms reported above demonstrate the tradeoffs between energy and delay that arise with any simple threshold-based technique. The differences between the Powerbook and HP-UX traces show how important the workload is in this regard: for the HP-UX trace on the Go•Drive disk, a 1-second threshold performed best out of this class of algorithms, reducing power consumption 45% (within 7% of optimal) compared to the 5-minute threshold, while for the Powerbook traces a 10-second threshold was better (and even then reduced power by only 11%). This is not surprising given the difference in mean interarrival times described in Section 4.1.

Regardless of the subtle differences in energy consumption between relatively short thresholds of 1, 2, or 10 seconds, it is clear that any of these short thresholds is much more energy efficient than the manufacturers' commonly recommended spin-down threshold of 5 minutes. However, the shorter thresholds introduce more spin-up delays. As a result, the most energy efficient threshold may not be the most desirable one. For example, with the Powerbook trace running on the Kittyhawk, moving from a threshold of 1s to 5s increases energy consumption by 16% but reduces the read spin-up delays by 42%.

5.3 Predictive Algorithms

We experimented with heuristics for predicting when to spin down based on past history rather than just the time since the last access. In order to ensure that a bad prediction did not result in keeping the disk spinning indefinitely, we used a threshold as a fallback: if the simulator predicted that an access would occur quickly enough that the disk should not spin down, and the time between accesses passed the threshold value, the disk would then spin down anyway (just like `THRESHOLD_DEMAND`). To date our results have been disappointing, with the energy consumption and response time from the predictive algorithms generally being slightly worse than the corresponding `THRESHOLD_DEMAND` algorithm.

The one case in which `PREDICTIVE_DEMAND` out-performed `THRESHOLD_DEMAND` was when the Powerbook trace was simulated with no buffer cache. In that case, the energy consumption of `PREDICTIVE_DEMAND` was roughly halfway between the optimal energy consumption and the best `THRESHOLD_DEMAND` policy, but all of these were much higher than the energy consumed using a moderate-sized buffer cache. Our conclusion with respect to `PREDICTIVE_DEMAND` is that the buffer cache tends to increase the entropy of interarrival times as seen by the disk; without a cache, access patterns are more predictable.

We have not yet experimented enough with `PREDICTIVE_PREDICTIVE` to be able to comment on it definitively. Our impression so far is that predicting the next access time well enough to spin up the disk just ahead of it will be very difficult, and the penalty for predicting incorrectly (spinning up uselessly, then spinning down again, or not spinning up when needed) will outweigh the benefits from "guessing" correctly. However, the area does bear further exploration.

5.4 The Impact of Disk Characteristics

The figures above show both the Kittyhawk and the Go•Drive drives using the same sets of traces and spin-down parameters. The lower operating costs for the Kittyhawk result in less power consumed for the same policies, and the faster and less power-intensive spin-up for the Kittyhawk makes it more feasible to quickly spin down the disk.

Since the Go•Drive consumes more power than the Kittyhawk when operating, and takes much longer to spin up and spin down, its overall power consumption is far greater than the same workload on the Kittyhawk. The impact of varying the spin-down threshold is less than for the Kittyhawk as well. Of course, operations on larger disk drives may be expected to consume more power than those on small ones, and spinning them up is especially costly—both in current and time—by comparison. For the foreseeable future, if users wish to store more data on their mobile computer they must be prepared for shorter battery life, poorer response time, or both.

6 Related Work

Li, *et al.*, have also investigated the issue of disk drive power management [12]. They used trace-driven simulation to look at a `THRESHOLD_DEMAND` policy for disk spin-control, and studied important buffer cache parameters. There are three important differences between our work and theirs. The first difference is that we consider multiple algorithms for determining when to spin down the disk: off-line optimal, `THRESHOLD_DEMAND`, and predictive. In contrast, they focused on `THRESHOLD_DEMAND`. The second difference is that they did a more detailed analysis of the impact of the buffer cache on power consumption and performance. We simulated the buffer cache only for the Powerbook trace and did not study the impact of different amounts of cache. We did not simulate a buffer cache at all for the HP trace since it is traffic filtered by the HP-UX buffer cache prior to being measured. This filtering made it impossible to experiment with the buffer cache. Finally, they considered the impact of spin-downs on disk reliability; we do not address this issue, relying on their prediction that disk-drive manufacturers will greatly increase the number of spin-up/spin-down cycles a disk can tolerate.

Their basic conclusion is the same as ours: short timeouts on the order of a few seconds greatly reduce power consumption by the disk and do not significantly degrade performance. Of course, there are differences in the exact amount of power saved and the impact on performance. They report that almost 90% of disk power consumption can be eliminated. Our results are less optimistic, with power consumption being reduced by 35–50%. They measure the impact on performance differently from us, making it somewhat difficult to compare results. They found that a 2-second timeout caused 15–30 seconds of delay per hour. With the 4-hour Powerbook trace, a 1-Mbyte buffer cache, and a Kittyhawk disk, we found that 388 read accesses had to wait for spin-up. Since in our simulation the Kittyhawk went immediately from the spin-down state to the halt state, spinning up would take approximately 1.5s, for a total of 153 seconds of delay per hour. This is not surprising, since the mean interarrival time in the Powerbook trace was less than a second. With the HP-UX trace, there were about 840 delays over 168 hours for an average delay of 7.5 seconds per hour. This large difference from the Powerbook simulation is due to the long periods of

inactivity during the HP-UX trace.

The other differences in our results reflect the traces used to drive the simulation and the disks that were simulated. Like us they used traces from Unix workstations: in particular the Sprite traces from the recent Berkeley study [7]. Unlike us they used traces from DOS PCs. They simulated a Maxtor drive; we simulated an HP Kittyhawk drive and a Quantum Go•Drive.

As mentioned above in Section 3.3, John Wilkes at Hewlett-Packard proposed a predictive algorithm for disk management [17]. He suggested adjusting spin-down timeouts based on a weighted average of recent interarrival times. Picking the weights may be a difficult task: we attempted to implement this strategy but were unable to out-perform THRESHOLD_DEMAND with the particular algorithms we tried. To the best of our knowledge, no other implementation of this strategy has been attempted.

In other work, Greenawalt [8] did an analytic study of disk management strategies. He assumed a Poisson process for request arrival; this is a questionable assumption given the clustering that tends to occur in real workloads. He considered two synthetic workloads, depending on the interarrival rate. He defined the "critical rate" as the number of accesses per unit time at which it is more power efficient to leave the disk spinning than to spin it down. His analysis is useful as an off-line policy; an on-line policy must be able to respond to and anticipate changes in the request arrival rate, something not addressed in this paper. Finally, like Li, *et al.*, Greenawalt studied the impact of the spin-down timeout on the reliability of the drive, an issue which we do not consider.

Finally, some commercial products have recently begun to address this issue. For example, while the standard Apple Powerbook control panel only allows the user to choose broadly between "maximum conservation" and "maximum performance," the Connectix Powerbook Utilities (CPU) [4] provide fine-grained control over such details as the disk spin-down threshold and processor speed, as well as feedback on the current state of the disk (such as a count-down to when the disk will spin down). Personal experience with CPU shows that a short spin-down delay, on the order of several seconds, does extend battery life at the cost of increased disk spin-ups and correspondingly slow response. The poor response time was sufficient to justify increasing the spin-down threshold from 5s to 15s in order to make the execution environment acceptable.

7 Conclusions and Future Work

We have simulated techniques for minimizing the power consumption of hard disks on mobile computers by spinning the disk only when necessary. Our off-line policy, OPTIMAL_OPTIMAL, demonstrates that significant power savings are possible: it can reduce disk power consumption by 35–50%, compared to the fixed threshold suggested by manufacturers. Our THRESHOLD_DEMAND policy demonstrates that it is possible for practical policies to get power savings close to that of the off-line policy. Threshold policies that spin down the disk after 1–10 seconds come within 7–27% of the off-line policy, and consume only 56–86% of the energy consumed by manufacturers' recommended thresholds (5 seconds for the Kittyhawk, and 5 minutes for the Go•Drive 120).

Threshold policies with fast spin-down save energy but degrade response time. We do not yet have enough experience to know what the best metric of degradation might be: in this paper we have compared algorithms based on the number of times a read request is delayed, but as noted above, other metrics are possible.

Finally, the impact of hardware design on software parameters is significant. The Kittyhawk was designed to make the transition between the active and idle states much less prohibitive than the Go•Drive or comparable disk drives. As a result, a shorter threshold is appropriate for the Kittyhawk than for these other disks, though even the Kittyhawk is susceptible to workloads that will result in too much overhead to make frequent spin-downs feasible. Ultimately a better approach than very short spin-down timeouts may be predictive algorithms that exploit past history, but such predictive algorithms may prove to be elusive.

Acknowledgements

We are grateful to John Wilkes and Hewlett-Packard Company for making their file system traces available to us. We also thank Brian Bershad, Ramón Cáceres, Hank Korth, and Kai Li, who have provided helpful feedback throughout our research; Bill Sproule, who collected the Powerbook traces and provided information about the Macintosh; and Bruce Zenel, who performed the measurements of power consumption reported in Section 2. Lastly, we thank the USENIX referees for their comments.

References

- [1] AppleLink. Developer Support:Developer Talk. AppleLink bulletin board, October 1993.
- [2] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-volatile memory for fast, reliable file systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 10–22, Boston, MA, October 1992. ACM.
- [3] Ramón Cáceres, Fred Douglass, Kai Li, and Brian Marsh. Operating Systems Implications of Solid-State Mobile Computers. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 21–27. IEEE, October 1993.
- [4] Connectix Corporation, San Mateo, CA. *CPU Connectix PowerBook Utilities Version 2.0 Addendum*, April 1993.
- [5] Dell Computer Corporation. *Dell System 320SLi User's Guide*, June 1992.
- [6] Intel Corporation/Microsoft Corporation. *APM Specification Version 1.0*.
- [7] Mary Baker et al. Measurements of a distributed file system. In *Proceedings of the 13th Symposium on Operating System Principles*, pages 198–212, Pacific Grove, CA, October 1991. ACM.
- [8] Paul Greenawalt. Modeling Power Management for Hard Disks. In *Proceedings of the Symposium on Modeling and Simulation of Computer and Telecommunication Systems*, 1994. To appear.
- [9] Hewlett-Packard. *Kittyhawk HP C3013A/C3014A Personal Storage Modules Technical Reference Manual*, March 1993. HP Part No. 5961-4343.
- [10] Hewlett-Packard. Kittyhawk power management modes. Internal document, April 1993.
- [11] Hewlett-Packard Company, Corvallis Division, Corvallis, OR. *Omnibook 300 Operating Guide*, 1 edition, April 1993.
- [12] Kester Li, Roger Kumpf, Paul Horton, and Thomas Anderson. A Quantitative Analysis of Disk Drive Power Management in Portable Computers. In *Proceedings of the 1994 Winter USENIX*, 1994. To appear.
- [13] Brian Marsh, Fred Douglass, and P. Krishnan. Flash Memory File Caching for Mobile Computers. In *Proceedings of the 27th Hawaii Conference on Systems Sciences*. IEEE, 1994. To appear.
- [14] Quantum. *Go•Drive 60/120S Product Manual*, May 1992.
- [15] Chris Ruemmler and John Wilkes. UNIX disk access patterns. In *Proceedings of the Winter 1993 USENIX Conference*, pages 405–420, San Diego, January 1993.
- [16] Seagate Technology. *Datasheet for Seagate ST7XXA Family: 1.8-Inch IDE FlashDrives*. Scotts Valley, CA, 1993.
- [17] John Wilkes. Predictive power conservation. Technical Report HPL-CSP-92-5, Hewlett-Packard Laboratories, February 1992.
- [18] Zenith Data Systems, Groupe Bull. *MastersPort 386SL/386SLe Owners Manual*, 1991.

Trademarks

Powerbook is a registered trademark of the Apple Computer Corporation. DECstation is a registered trademark of Digital Equipment Corporation. Kittyhawk and HP-UX are registered trademarks of Hewlett-Packard Company. Word is a registered trademark of Microsoft Corporation. Eudora is a registered trademark of Qualcomm. GoDrive is a registered trademark of Quantum. UNIX is a registered trademark of UNIX System Laboratories, Inc.

Author Information

Fred Douglass is a research scientist at the Matsushita Information Technology Laboratory. His research interests include mobile and distributed computing, file systems, and user interfaces. He received a B.S. in Computer Science from Yale University in 1984. He received his M.S. and Ph.D. degrees in Computer Science from the University of California, Berkeley in 1987 and 1990. Reach him electronically at douglass@research.panasonic.com. Reach him via USMail at Matsushita Information Technology Laboratory; 2 Research Way, Third Floor; Princeton, NJ 08540-6628.

P. Krishnan is a Ph.D. student at Brown University. His research interests include the development and analysis of algorithms, online and prediction algorithms, prefetching and caching, mobile computing, data compression, databases systems and operating systems. He received his B. Tech in Computer Science and Engineering from the Indian Institute of Technology, Delhi, in 1989, and his Masters in Computer Science at Brown University in 1991. He is currently visiting Duke University. Reach him electronically at pk@cs.brown.edu. Reach him via USMail at Box 90129, Duke University; Durham, NC 27708-0129.

Brian Marsh is a research scientist at the Matsushita Information Technology Laboratory. His research interests include operating systems, particularly for mobile computers. He received an A.B. in Computer Science from the University of California, Berkeley in 1985. He received his M.S. and Ph.D. degrees in Computer Science from the University of Rochester in 1988 and 1991. Reach him electronically at marsh@research.panasonic.com. Reach him via USMail at Matsushita Information Technology Laboratory; 2 Research Way, Third Floor; Princeton, NJ 08540-6628.

A Usage Profile and Evaluation of a Wide-Area Distributed File System *

Mirjana Spasojevic
Transarc Corporation

M. Satyanarayanan
Carnegie Mellon University

Abstract

The evolution of the Andrew File System (AFS) into a wide-area distributed file system has encouraged collaboration and information dissemination on a much broader scale than ever before. In this paper, we examine AFS as a provider of wide-area file services to over 80 organizations around the world. We discuss usage characteristics of AFS derived from empirical measurements of the system, and from user responses to a questionnaire. Our observations indicate that AFS provides robust and efficient data access in its current configuration, thus confirming its viability as a design point for wide-area distributed file systems.

1. Introduction

Over the last decade, distributed file systems such as AFS and NFS in the Unix world, and Netware and LanManager in the MS-DOS world have risen to prominence. Today, virtually every organization with a large collection of personal machines uses such a system. The stunning success of the distributed file system paradigm is attributable to three factors.

First, a distributed file system simplifies the *separation of administrative concerns* from usage concerns. Users work on tasks directly relevant to them on their personal machines. Incidental but essential tasks such as backup, disaster recovery, and expansion of disk capacity are handled by a professional staff who focus primarily on the servers.

Second, the use of a distributed file system simplifies the *sharing of data* within a user community. Such sharing can arise in two forms: by a user accessing his files from different machines, and by one user accessing the files of another user. The ability to easily access one's files from any machine enhances a user's mobility within his organization. Although the accessing of someone else's files is not a frequent event (a fact confirmed by many previous studies [1, 6]), ease of access once the need arises is perceived as a major benefit by users. In other words, while sharing may be rare, the payoff of being able to share easily is very high¹.

Third, *transparency* is preserved from the users' and applications' points of view. Applications do not have to be modified to use a distributed file system. Because a distributed file system looks just like a local file system, a user does not have to learn a completely new set of commands or new methods of file usage.

The designs of modern distributed file systems reflect these observations. They use a client-server model, offer location transparency, rely on caching to exploit locality, provide fairly weak consistency semantics

*This research was funded by the Advanced Research Project Agency, under contract number MDA972-90-C-0036, ARPA order number 7312. The views and conclusions expressed in this paper are those of the authors and do not represent the official position of ARPA, Transarc Corporation or Carnegie Mellon University. Please direct correspondence to Mirjana Spasojevic, Transarc Corporation, The Gulf Tower, 707 Grant Street, Pittsburgh, PA 15219.

¹In this respect a distributed file system is like a telephone system: although a given individual only tends to call a tiny fraction of all telephone numbers, the latent ability to effortlessly reach any other telephone in the world is viewed as a major asset of the system.

relative to databases, and support programming and user interfaces that are close to those of a local file system. The success and widespread usage of these systems confirms the appropriateness of these design choices.

But this success engenders a new question: "Is the distributed file system paradigm sustainable at very large scale?" In other words, how well can a very large distributed file system meet the goals of simplifying system administration, supporting effective sharing of data, and preserving transparency? Growth brings many problems with it [12]: the level of trust between users is lowered; failures tend to be more frequent; administrative coordination is more difficult; performance is degraded. Overall, mechanisms that work well at small scale tend to function less effectively as a system grows. Given these concerns, how large can a distributed file system get before it proves too unwieldy to be effective?

In this paper, we seek to answer this question by studying the usage characteristics of AFS, the largest currently deployed instance of a distributed file system. At the time of writing, AFS unites about 1,000 servers and 20,000 clients in 7 countries into a single file name space. We estimate that more than 100,000 users use this system worldwide. In geographic span as well as in number of users and machines, AFS is the largest distributed file system that has ever been built and put to serious use.

Our study confirms that the distributed file system paradigm is indeed being effectively supported at the current scale of AFS. Further, our data does not expose any obvious impediments to further growth of the system. While asymptotic limits to growth are inevitable, they do not appear to be just around the corner.

2. AFS Background

The rationale, detailed design, and evolution of AFS have been well documented in previous papers [2, 5, 9, 10, 11, 15]. In this section, we only provide enough details of the current version of AFS (AFS-3) to make the rest of the paper understandable.

Using a set of trusted servers, AFS presents a location-transparent Unix file name space to clients. Files and directories are cached on the local disks of clients using a consistency mechanism based on *callbacks* [3]. Directories are cached in their entirety, while files are cached in 64 KB chunks. All updates to a file are propagated to its server upon *close*. Directory modifications are propagated immediately.

Backup, disk quota enforcement, and most other administrative operations in AFS operate on *volumes* [13]. A volume is a set of files and directories located on one server and forming a partial subtree of the shared name space. A typical installation has one volume per user, one or more volumes per project, and a number of volumes containing system software. The distribution of these volumes across servers is an administrative decision. Volumes that are frequently read but rarely modified (such as system binaries) may have read-only replicas at multiple servers to enhance availability and to evenly distribute server load.

AFS uses an *access list* mechanism for protection. The granularity of protection is an entire directory rather than individual files. Users may be members of *groups*, and access lists may specify rights for users and groups. Authentication relies on *Kerberos* [16].

AFS supports multiple administrative *cells*, each with its own servers, clients, system administrators and users. Each cell is a completely autonomous environment. But a federation of cells can cooperate in presenting users with a uniform, seamless file name space. The ability to decompose a distributed system into cells simplifies delegation of administrative responsibility [15].

As originally designed, AFS was intended for a LAN. However, the RPC protocol currently used in AFS has been designed to perform well both on LANs as well as on wide-area networks. In conjunction with the cell mechanism, this has made possible shared access to a common, world-wide file system distributed over nodes in many countries.

In 1990 the Advanced Research Projects Agency (ARPA) awarded Transarc a contract to deploy and

evaluate a file system to be shared by 40 to 50 Internet sites in the US. By mid-1991 there were 14 organizations included in the study. At the time of writing this paper, more than 80 organizations were part of this wide-area distributed file system (*wadfs*).

The wide-area nature of AFS is clearly visible from Figure 1, which shows the cells visible at the topmost level of AFS. All these directories, as well as the trees beneath them, are accessible via normal Unix file operations to any workstation anywhere in the system.

3. Evaluation Methodology

A comprehensive characterization of this system would include an assessment of basic architectural features, an analysis of quantitative data from the deployed system, and an examination of qualitative information reflecting on issues such as user perceptions of quality.

Since earlier papers have explored the architecture of AFS in detail, we omit it from this paper. Here we report on AFS from two angles: first, by instrumenting clients and servers and collecting data over a period of time; second, by circulating a questionnaire on various aspects of AFS to a sample of users and summarizing their responses. We believe that this combination of quantitative and qualitative information fairly characterizes the current state of the system.

One's confidence in the answers of an evaluation can be classified into four levels based on the origin of the information: *intrinsic* (direct examination of the system design), *empirical* (raw measurements), *evidentiary* (inferences based on raw data), and *anecdotal* (information requiring user judgment). In this taxonomy, our quantitative information is empirical and evidentiary while our qualitative information is anecdotal.

3.1. Quantitative Data

Empirical measurements of AFS were performed through the *xstat* data collection facility [17]. The AFS code was instrumented to allow collection of extended statistics concerning the operation of servers and clients. These statistics could be obtained remotely via an RPC call. A central data collection machine, located at Transarc, polled and obtained data from each participating machine four times a day. The collected data was formatted and inserted into a relational database for postprocessing. Figure 2 shows the structure of our data collection mechanism.

The scale of the system complicated the logistics of data collection considerably. It would have been practically infeasible to require the active cooperation of users or system administrators at many different cells to assist in the data collection. Hence our instrumentation required no regular administrative effort by the sites being monitored. However, the system administrator of a cell could turn off data gathering if that cell did not wish to participate in the study.

Not requiring the active cooperation of remote cells complicated the process of discovering which clients and servers should be contacted for data collection. Our solution to this problem was to run a discovery process once every few weeks. This process queried the Domain Name Service at each cell to obtain a list of registered IP addresses. This list was then probed to discover new AFS clients and servers in that cell.

The measurements were conducted during a 12-week data collection period from mid-May to mid-August 1993. Our data spans 50 file servers and 300 clients from 12 cells in 7 states. The only factors limiting broader coverage were the deadlines for this paper, and the need for participating sites to pick up the versions of AFS software incorporating our instrumentation.

cs.arizona.edu	theory.cornell.edu	soup.mit.edu	spc.uchicago.edu
cs.brown.edu	kiewit.dartmouth.edu	watch.mit.edu	ucop.edu
bu.edu	northstar.dartmouth.edu	ncat.edu	ni.umd.edu
cmu.edu	iastate.edu	eos.ncsu.edu	wam.umd.edu
andrew.cmu.edu	ucs.indiana.edu	nd.edu	umich.edu
club.cc.cmu.edu	isi.edu	nsf-centers.edu	citi.umich.edu
ce.cmu.edu	alefnull.mit.edu	pitt.edu	math.lsa.umich.edu
cs.cmu.edu	athena.mit.edu	psc.edu	lsa.umich.edu
ece.cmu.edu	rel-eng.athena.mit.edu	rose-hulman.edu	cs.unc.edu
sei.cmu.edu	media-lab.mit.edu	rpi.edu	css.cs.utah.edu
cs.cornell.edu	net.mit.edu	dsg.stanford.edu	cs.washington.edu
graphics.cornell.edu	sipb.mit.edu	ir.stanford.edu	

(a) educational cells

ads.com	ctp.se.ibm.com	prc.unisys.com	gr.osf.org
bstars.com	mtxinu.com	stars.reston.unisys.com	ri.osf.org
cards.com	locus.com	grand.central.org	syseng.osf.org
pub.nsa.hp.com	stars.com	ciesin.org	
palo_alto.hpl.hp.com	transarc.com	dce.osf.org	

(b) commercial cells

inel.gov	alw.nih.gov	ssc.gov
nersc.gov	ctd.ornl.gov	cmf.nrl.navy.mil

(c) government cells

jrc.flinders.oz.au	uni-freiburg.de	etl.go.jp	pegasus.cranfield.ac.uk
glade.yorku.ca	rus.uni-stuttgart.de	others.chalmers.se	athena.ox.ac.uk
writer.yorku.ca	sfc.keio.ac.jp	nada.kth.se	
lrz-muenchen.de	titech.ac.jp	bcc.ac.uk	

(d) cells outside US

This figure shows the cells visible from a typical client in the system. The listing above was obtained by doing an "ls /afs" and then sorting the output according to the domain. As the figure shows, there are 47 educational cells, 18 commercial, 6 governmental, and 14 cells outside the United States.

Figure 1: Cells visible from a typical AFS client.

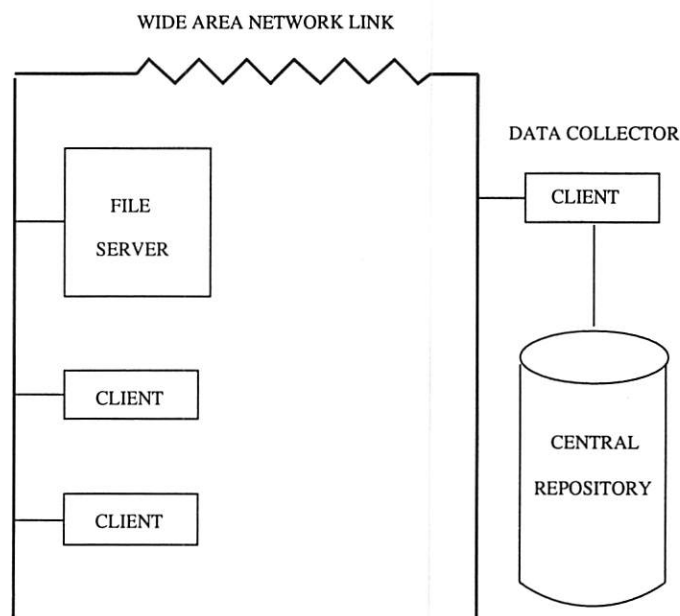


Figure 2: Instrumentation for Data Collection

3.2. Qualitative Data

To complement the quantitative data obtained by instrumentation, we constructed a questionnaire that touched upon a diverse set of issues. The purpose of the questionnaire was to elicit user perceptions as well as to obtain a profile of AFS usage. The topics of interest to us included characterization of the user community, extent of usage of native and foreign cells, and degree of collaboration within and across cells. We were also interested in obtaining user perceptions of performance and reliability of AFS for native and foreign cell access. Finally, we were interested in the value and adequacy of various AFS mechanisms such as access control lists, read-only replication, and data mobility.

The questionnaire was distributed in two ways: first, by posting on several Netnews bboards; second, by direct mailing to AFS contacts in different cells. We received about 100 responses from 50 cells. The data we present in this paper is averaged over all these responses.

4. Observations and Analysis

In this section we present both quantitative and qualitative data collected during our 12-week study. We begin by examining storage capacity and user profile. We then discuss the nature of client-server interaction, including RPC traffic and bulk data transfers. Next, we explore cache performance and availability, two key parameters of any distributed system. Finally, we examine the extent to which AFS is used for collaboration and information dissemination. In discussing these issues, we interleave the results of both empirical and anecdotal evidence, pointing out corroborations and contradictions wherever appropriate.

4.1. AFS Usage

4.1.1. Data Profile

Table 1 shows a recent snapshot of the data stored at 17 cells². These cells comprised 95 file servers, housing almost 50,000 volumes and constituting over 300 GB of data. The data shows that although over

²These 17 cells were a superset of the 12 from which all other statistics in this paper are reported. We were able to obtain a larger sample in this case because the necessary instrumentation was present in an earlier release of AFS.

Volume type	Total	Size (GB)	Avg (MB/vol)
User	25,630	73	2.9
Backup	14,557	105	7.2
Readonly	2,121	24	11.4
Other	7,595	111	14.6
ALL	49,903	313	6.3

Table 1: Storage Capacities of 17 Cells

50% of the volumes belong to individual users, they contain only 23% (73 GB) of the data. A third of the data (over 100 GB) belongs to backup volumes. Only 4.2% of the volumes are readonly replicas, and they contain only 7.7% of the data. The remaining 15% of the volumes correspond to system binaries and data, bulletin boards, and other miscellaneous data. Together, these volumes contain one third of the total data.

Extrapolating from this evidence, and from additional information from the questionnaire, we estimate that the whole wadfs contains more than 200,000 volumes with 1.5-2 TB of data. It is interesting to note that although the average volume size is only 6.3MB, the raw data indicates that some volumes contain more than 1.5GB of data. In other words, volumes span a wide range of sizes but tend to be skewed toward the low end.

A related but distinct question pertains to how many of these volumes are in active use every day. To answer this question, we recorded the number of volumes whose activity level exceeded a specified threshold each day for the duration of our data collection. The activity level was arbitrarily chosen to be 10 read references to a volume. Our data showed that, on average, a server has 65 active volumes, each containing about 16MB of data.

4.1.2. User Profile

The AFS user community consists of a number of academic, government and commercial sites and AFS users tend to have a very diverse background. However, responses to our questionnaire came mostly from AFS contacts, who are usually system administrators (Figure 3)³. The majority of respondents use AFS daily and for most of them the typical AFS session lasts a full working day. Most of them are serious programmers and two-thirds of them rate their knowledge of AFS to be at an advanced or expert level. Most of them had experience with other distributed file systems, usually NFS. Our sample thus represents a technically sophisticated group of respondents. This renders their assessments of AFS quality more credible, but also leaves unanswered the question of how naive users view AFS.

4.2. Client-Server Interaction Profile

How do AFS clients and servers interact? The answer to this question is important because knowledge of the relative distribution of file system RPC calls helps characterize a normal system and identifies the most common calls. This, in turn, allows performance tuning to be focused. Figure 4 lists the client-server RPC calls with short descriptions.

Both servers and clients have been instrumented to record the information regarding these calls. They keep statistics about the total number of calls, the number of successful calls and the average time of execution of successful calls (with the standard deviation). During our study, statistics were collected from 46 file servers and 264 clients on a typical day.

³The percentages for some questions do not add up to 100% because some respondents did not answer particular questions or they marked more than one choice.

1. What is your occupation?
 - 9% Student
 - 16% Researcher/Scientist
 - 32% Software Developer
 - 8% Manager
 - 11% Support Staff
 - 49% System Administrator
 - 2% Other
2. How often do you typically use AFS?
 - 0% Never
 - 1% Rarely
 - 11% Periodically
 - 85% Daily
3. How long does your typical AFS session last?
 - 6% Under 30 min
 - 5% 30 min to 1 hr
 - 12% 1 to 3 hr
 - 74% Full working day
4. Which best describes the depth of your general computing experience?
 - 1% Novice
 - 15% Casual programmer
 - 81% Serious programmer
 - 2% Non-technical user
5. How would you rate your knowledge of AFS?
 - 2% Novice
 - 28% Intermediate
 - 50% Advanced intermediate
 - 17% Expert
6. What other distributed file systems have you worked with?
 - 89% NFS
 - 17% Apollo Domain
 - 7% RFS
 - 9% Other(s)
7. What's the best description of how you use AFS with the other file service resources at your site?
 - 0% Don't use AFS at all
 - 5% Use existing files in AFS, but none of my files are there
 - 13% Store some of my files in AFS, most on other systems
 - 14% Store many of my files in AFS
 - 66% Most of my files are in AFS, including my home directory

Figure 3: A Profile of Survey Participants

Fetch_Data	Returns data of the specified file or directory and places a callback on it.
Fetch_ACL	Returns the content of the specified file's or directory's access control list.
Fetch_Status	Returns the status of the specified file or directory and places a callback on it.
Store_Data	Stores data of the specified file or directory and updates the callback.
Store_ACL	Stores the content of the specified file's or directory's access control list.
Store_Status	Stores the status of the specified file or directory and updates the callback.
Remove_File	Deletes the specified file.
Create_File	Creates a new file and places a callback on it.
Rename	Changes the name of a file or directory.
Symlink	Creates a symbolic link to a file or directory.
Link	Creates a hard link to a file.
Make_Dir	Creates a new directory.
Remove_Dir	Deletes the specified directory which must be empty.
Set_Lock	Locks the specified file or directory.
Extend_Lock	Extends a lock on the specified file or directory.
Release_Lock	Unlocks the specified file or directory.
GiveUp_Call	Specifies a file that a cache manages has flushed from its cache.
Get_Vol_Info	Returns the name(s) of servers that store the specified volume.
Get_Vol_Status	Returns the status information about the specified volume.
Set_Vol_Status	Modifies status information on the specified volume
Get_Time	Synchronizes the workstation clock and checks if servers are alive.
Bulk_Status	Same as Fetch_Status but for a list of files or directories.

Figure 4: Client-Server RPC Calls

Type of call	%	# of calls	(% err.)	Avg ms	(s.d.)
1. Fetch_Data	7.6	33,427,405	(0.2)	116	(486)
2. Fetch_Status	67.0	295,247,833	(18.0)	12	(378)
3. Store_Data	4.0	17,336,400	(1.0)	157	(744)
4. Store_Status	8.7	38,399,197	(0.3)	3	(119)
5. Remove_File	1.9	8,172,106	(0.0)	40	(335)
6. Create_File	2.0	8,945,032	(15.7)	22	(545)
7. Extend_Lock	1.8	7,815,294	(73.1)	9	(291)
8. GiveUp_Call	1.6	6,839,076	(0.0)	1	(39)
9. Get_Time	3.2	14,210,834	(0.0)	4	(800)
ALL	100.0	440,778,197	(13.8)	n/a	

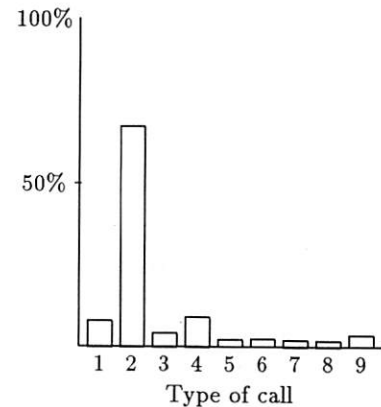


Table 2: Average Distribution of RPC Calls Observed by Servers

4.2.1. RPC Calls Observed by Servers

Over 440 million calls were observed during the data collection period. About 86% of these were successful. Table 2 summarizes the detailed statistics of calls accounting for at least 1% of the total.

The most frequent is **Fetch_Status** call. We conjecture that many of these calls are generated by users listing directories in parts of the file name space that they do not have cached. The relatively high number of unsuccessful calls (18%) suggests that these directories belong to some protected areas of the file name space. It is interesting to note that despite caching, the number of **Fetch_Data** calls is considerably higher than the number of **Store_Data** calls. Both **Fetch_Data** and **Store_Data** calls take considerably longer than other operations. This is to be expected, since they involve disk I/O.

GiveUP_Call turned out to be the call that takes the least amount of time on average. It was even faster than the **Get_Time** call, which is the simplest call. Considering the very high standard deviation of the **Get_Time** call, this might be just an anomaly in the collected data, but it can also be the result of a slow system call to get the time.

Although **Fetch_ACL** is not shown in Table 2, our raw data showed that it takes considerably more time on average than **Fetch_Status**. This surprised us, since **Fetch_Status** returns access list information. This apparent anomaly was explained when inspection of the AFS code showed that the implementation of **Fetch_ACL** contains a call to a protection server, while the implementation of **Fetch_Status** does not.

Analysis of RPC calls on a weekly basis confirms that their distribution is stable over time. Table 3 presents this data. This data shows only two significant deviations from the general profile shown in Table 2. One anomaly is the very high number of **Store_Status** calls during weeks 10 and 11. We discovered that more than 90% of these calls were concentrated on three file servers at Transarc. Further investigation revealed that these servers are frequently used for testing new AFS releases, thus explaining the unusual distribution of calls.

The second anomaly is the unusually high number of **Extend_Lock** calls during week 4. This is usually a rarely-occurring call, typically accounting for less than 1% of the calls in other weeks. Detailed analysis of week 4's data showed that the majority of these **Extend_Lock** calls were concentrated on just one file server. Our hypothesis is that there was a orphaned process on one of the clients repeatedly trying to make an **Extend_Lock** call, but failing because of expired authentication tickets. This also explains the high percentage of failed **Extend_Lock** calls in Table 2.

Based on this data, one can loosely characterize a normally running system as one with a very high

week	Fetch_D	Fetch_S	Store_D	Store_S	Remove_F	Create_F	Extend_L	GiveUp_C	Get_T
1	8.4	73.2	3.2	2.0	1.2	1.4	3.1	1.8	4.0
2	8.1	71.5	3.4	4.9	1.5	1.6	1.0	1.8	3.9
3	8.2	71.6	3.6	3.7	1.3	1.7	2.2	1.5	4.4
4	7.5	62.3	3.5	4.7	1.4	1.7	12.0	1.3	3.5
5	7.1	76.9	3.3	2.4	1.2	1.6	0.5	1.4	3.7
6	7.3	70.9	4.0	6.3	2.2	2.4	0.3	1.4	2.6
7	7.4	71.0	4.0	5.8	1.7	2.2	0.5	1.8	3.6
8	8.7	66.7	4.2	7.3	2.0	2.5	0.4	1.6	2.8
9	7.1	72.9	3.3	6.4	1.5	1.6	0.4	1.6	3.1
10	7.3	53.6	4.8	21.1	2.8	2.4	0.3	1.3	3.2
11	7.2	52.9	5.4	22.1	2.8	2.6	0.4	1.3	2.5
12	7.0	74.5	3.2	6.1	1.2	1.6	0.6	1.9	2.5
all	7.6	67.0	4.0	8.7	1.8	2.0	1.8	1.5	3.2

This table is based on the same raw data as Table 2. It indicates weekly averages (in percentages), rather than averaging across all weeks.

Table 3: Weekly RPC Call Distributions Observed by Servers

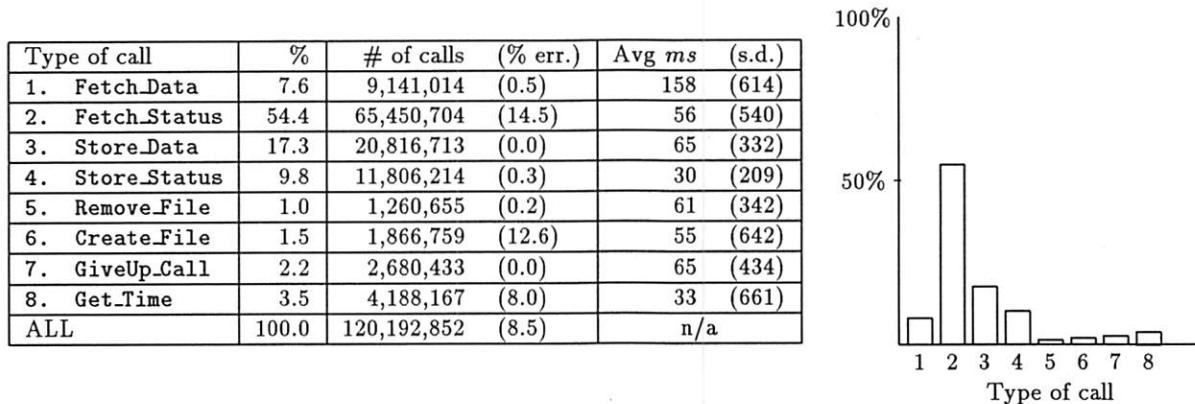


Table 4: Average Distribution of RPC Calls Generated by Clients

number (above 60%) of **Fetch.Status** calls, and smaller, but still significant, number of **Fetch.Data** and **Store.Status** calls (about 8%). Other frequent calls in such a system include **Store.Data** and **Get.Time**.

4.2.2. RPC Calls Generated by Clients

The set of machines from which we were collecting data did not represent a “closed system”, i.e. there was no guarantee that participating servers and clients were contacting only each other. Thus, the number of calls observed by file servers does not match the number of calls generated by clients. Nevertheless, it is interesting to compare these two profiles. Table 4 summarizes the data collected from clients.

There were over 120 million calls, out of which 91.5% were successful. Again, **Fetch.Status** calls dominate. But the relative percentage of these calls was significantly lower than that reported in Table 2 for servers. At the same time, the relative percentage of **Store.Data** calls was significantly higher. Examination of the raw data showed that most of **Store.Data** calls came from a set of eight machines belonging to one cell. We conjecture that the applications on these machines differed substantially from the norm in their

	Servers		Clients	
	Fetches	Stored	Fetches	Stored
0 B - 128 B	32 %	44 %	33 %	6 %
128 B - 1 KB	4 %	7 %	5 %	15 %
1 KB - 8 KB	43 %	14 %	37 %	26 %
8 KB - 16 KB	4 %	6 %	4 %	8 %
16 KB - 32 KB	2 %	4 %	3 %	7 %
32 KB - 128 KB	14 %	25 %	17 %	7 %
over 128 KB	1 %	0 %	0 %	0 %
Daily per machine	156 MB	116 MB	5.3 MB	4.7 MB

Table 5: File Transfer Size Distribution

file access patterns. When these machines are excluded from the data set, the frequency of `Fetch.Status` calls increases to 62% and the frequency of `Store.Data` calls drops to 5%. The frequencies of other calls are similar to those reported in Table 2.

Surprisingly, Table 4 shows the average `Store.Data` call to be much faster than the average `Fetch.Data` call. It is even faster than the average `Fetch.Data` call on servers (Table 2), indicating negative network delay! This anomaly is also caused by the above-mentioned group of eight clients. When they are excluded from the analysis, the average time of `Store.Data` calls increases to a more credible 149ms.

4.2.3. Causes of RPC Failures

As noted in the previous section, nearly 8.5% of the calls generated by clients failed. We were curious about the nature of these failures since they may have been symptomatic of underlying performance or reliability problems. To study this, AFS clients were instrumented to keep track of failed RPC calls. Errors were divided into several categories: server problems, network problems, protection problems (insufficient authorization or expired authorization tickets), volume problems, occurrences of a busy volume (e.g. when a volume is moved to another server) and errors of unknown cause.

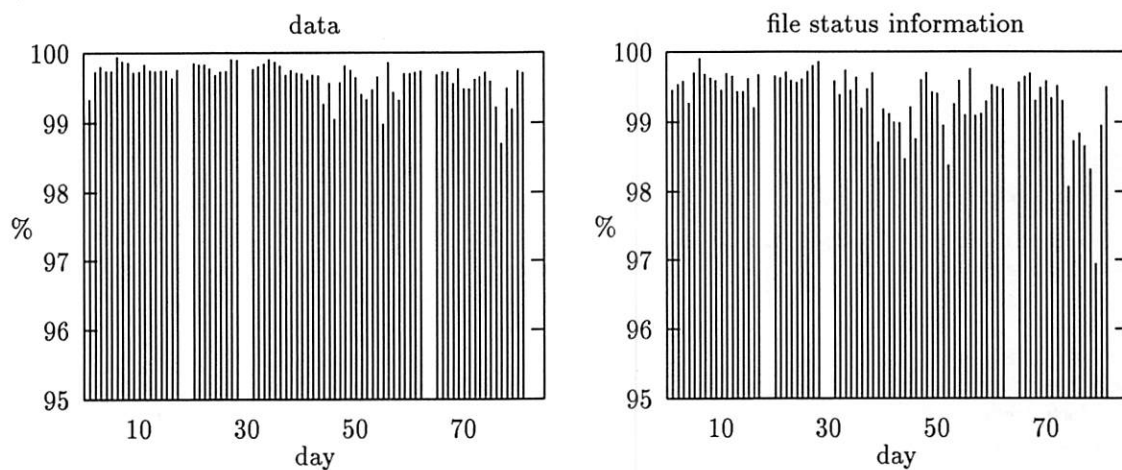
Our data showed that the majority of failed calls, 92%, were `Fetch.Status` calls. Most of them, 76%, failed because of protection errors. This is consistent with our earlier hypothesis of the existence of periodic jobs on some machines that attempt to traverse the AFS tree and fail when they encounter a protected part of the tree. Another plausible explanation is continuous execution of some background daemons (e.g. `xbiff`) which always produce a failed call after the authorization ticket's expiration. A significant number of unsuccessful calls, 22%, failed for unknown reasons.

4.2.4. Bulk Transfer Profile

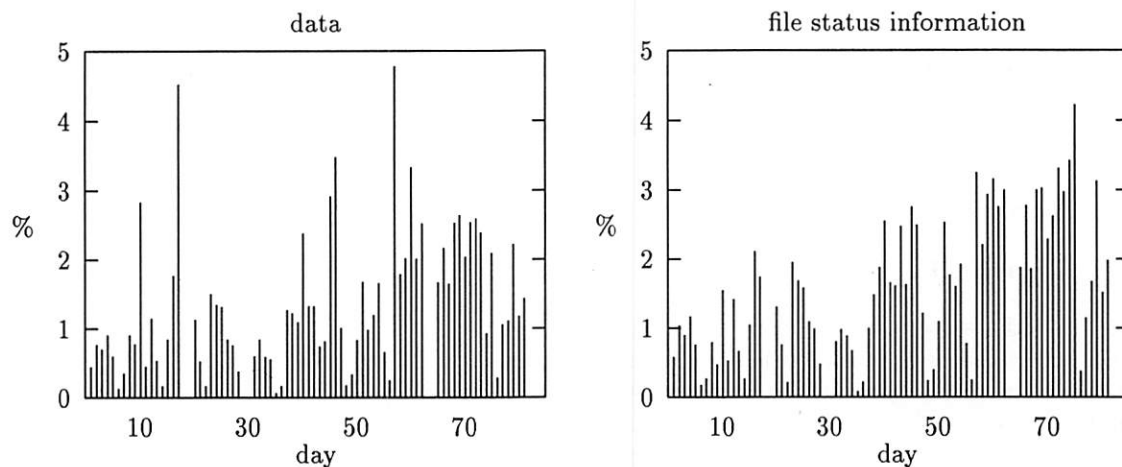
Statistics concerning file transfers were recorded by both file servers and clients. AFS performs partial file caching, so the numbers reported here show transfers on a per chunk basis, rather than on a per file basis. The exceptions are directories which are cached in their entirety. Chunk size is 64KB by default, but may be changed on a per-client basis.

The collected statistics are summarized in Table 5. Our data indicates that the most frequently fetched chunks are in the range 1-8KB. These correspond to entire files or directories. This result is consistent with many earlier studies of file size distributions which have reported small average file size [6, 8]. The second most frequently fetched chunk size is even smaller, in the range 0-128B.

The distribution of fetched data on file servers and clients is very similar. However, the distribution of stored data differs considerably. We can conclude that even when mixes of RPC calls and fetched data



(a) Combined cache hit ratio for native and foreign file references



(b) Fraction of references to files in foreign cells

This figure shows the observed cache hit ratios and relative proportion of native and foreign cell references over the data collection period. As explained in Section 4.3.1., data from six machines was excluded. Analysis of the raw data showed that the excluded machines exhibited comparable cache performance to the overall set of machines. The gaps in histograms on several days correspond to missing data due to problems with the data collection machine.

Figure 5: Cache Performance and Reference Mixes

8. How would you rate AFS performance when accessing files in your own cell (organization)?

22% Excellent
49% Good
20% Fair
4% Poor
3% Unsatisfactory

9. Compared to other distributed file systems you've used, is AFS in your own cell:

11% Much faster
30% Faster
32% Comparable
15% Slower
2% Much slower
5% Haven't used other distributed file systems

10. How would you rate AFS performance when accessing files in a remote cell?

8% Excellent
42% Good
38% Fair
6% Poor
2% Unsatisfactory
4% No experience

11. Have any of the following aspects of AFS ever seriously impeded your work?

65% Performance/reliability
21% Authentication/ACLs
6% Replication
19% Backup/restore
25% Semantics (Unix emulation)
32% Availability for other hardware/OS bases
10% Deployment (i.e., it doesn't run at the places with which I interact)
5% Other(s)

Figure 6: Users' Perception of AFS Performance

distributions are similar, there might be a significant variation in stored data distribution on servers and clients. The results from Section 4.1.1 indicate that the amount of data housed by active volumes is about 1GB per file server. Table 5 shows that only about 15% of this data (156MB) is actually fetched by clients.

4.3. AFS Performance

4.3.1. Cache Performance

Cache hit ratio is a critical factor in determining the overall performance of a system like AFS. Caching is especially valuable in masking the long latencies typical of wide-area networks. To study this aspect of AFS, clients were instrumented to keep statistics on cache hit rates and on the percentages of references made to native and foreign cells. Since the AFS file cache is split into a cache for data and a cache for status information, our statistics were kept separately for these two categories.

The overall percentage of references to remote files was 4.5% for data and 2.3% for status information. However, these numbers showed high variation from day to day: between 0.5% and 26% for data, and 0.5 and 34% for status. Closer inspection of the raw data revealed a group of six machines contributing to the majority of these references. We conjecture that these machines run periodic jobs that attempt to traverse the entire AFS tree⁴. Since these constitute pathological cases, we excluded these machines from our data set, and obtained the substantially more uniform results shown in Figure 5.

Our data indicates that the average cache hit ratio is over 98% for data and over 96% for status information. Over 95% of data and status references are to native cells. We statistically analyzed the possibility of foreign cell references causing much lower cache hit ratios. Our analysis indicated that there was no such correlation.

The responses to our questionnaire on AFS performance are presented in Figure 6. Most of the respondents rate the performance of AFS when accessing local data as good or excellent. Only 7% of users are not satisfied. AFS performance when accessing files in a remote cell is somewhat worse - 50% of respondents rate it as good or excellent, while 38% feel it is fair. Compared to other distributed systems they have used, 32% of respondents feel that AFS provides comparable performance, while 41% say that it is faster or much faster. Overall, the majority of users seem to be satisfied with AFS performance. But nearly two-thirds of

⁴This hypothesis has been verified for at least some of the machines.

Type of outage	% of time	Time (min/week)
Servers in the same cell	0.08 - 0.59%	1.2-8.5
Servers in the foreign cell	0.04 - 0.54%	0.5-7.7

This table shows observed average inconvenience times for clients over 12-week data collection period. The lower side of the range represents the case when for each client all daily failures occur simultaneously. The higher side of the range represents the case when daily failures do not overlap.

Table 6: Average Inconvenience Time for Clients

Downtime durations	Servers in the same cell	Servers in foreign cells
0 min - 10 min	1584	861
10 min - 30 min	759	128
30 min - 1 hr	484	67
1 hr - 2 hr	275	48
2 hr - 4 hr	140	21
4 hr - 8 hr	63	6
> 8 hr	44	28
TOTAL	3349	1159

Table 7: Distribution of File Servers Outage Durations

them also rate performance and reliability as aspects of AFS that have sometimes been unsatisfactory.

4.3.2. Frequency of File Server Failures

Interruption of file service in a wadfs is a potential obstacle to providing transparency. One way of measuring file server downtimes is to have file servers record downtimes themselves and report them to the data collection agents. However, in our view, a much more important picture is the one that client machines have about the file servers' availability. Thus, we instrumented clients to record outages. A particular file server's downtime was observed only by the clients that could not access particular data from that file server (because of the server's failure and/or network problems). Such an approach weights failures by clients' interest in the files affected; in other words, the inaccessibility of a heavily-used file contributes more to the metric than the inaccessibility of a lightly-used file. Table 6 reports average *inconvenience* time, which is the time during which a client cannot communicate with at least one file server that it needs to access.

Downtime incident statistics were collected from 235 clients on an average day. During the twelve week data collection period, the number of observed server downtime incidents was 3349 for servers in the same cell and 1159 for servers in foreign cells. (Table 7). It should be noted that a particular server's outage can be reported multiple times if observed by multiple clients. Also, on an average day only about 15% of the contacted clients accessed data in foreign cells and thus were able to observe server downtimes in foreign cells. According to the numbers collected, on average, a client observes a server outage every 5-6 days for the local cell and every 3-4 days for the foreign cell, under the assumption that all clients are equally observant (active). The duration of almost half the outages is less than 10min. Since this is shorter than the recovery time for a typical server, we conjecture that many of these short outages are really due to transient network failures.

Users tend to notice file server failures less frequently than what the empirical evidence indicates (Figure 7). Failures of servers in local cells are experienced at most once a month by 77% of respondents. Only 3%

12. In your experience, how often are the AFS File Servers in your own cell (organization) down or unavailable?

2% Never
36% Once every few months
39% Once a month
17% Once a week
3% Daily

13. In your experience, for how long are AFS File Servers typically down when they crash or in the presence of network problems?

13% Less than 10 minutes
36% 10 minutes to 30 minutes
35% 30 minutes to 1 hour
12% More than 1 hour
3% N/A

14. In your experience, how often are the AFS File Servers in other cells (organizations) you access down or unavailable?

6% Never
20% Once every few months
29% Once a month
17% Once a week
7% Daily
20% N/A

Figure 7: Users' Perception of File Server Failures

Cells contacted	% of clients
≥ 1	100
≥ 2	67
≥ 3	42
≥ 6	22
≥ 10	15
≥ 20	9
≥ 50	4
≥ 70	3

Table 8: Client Contacts with Cells

witness file server failures on a daily basis. However, users perceive failures as lasting longer than the empirical data indicates: less than 10min for 13%, 10 - 30 min for 36%, 30min - 1hr for 35%, and more than 1 hr for 12% of respondents. Failures of servers in foreign cells are experienced at most once a month by 54% of respondent. However, the actual percentage is higher, because this question did not apply to 20% of respondents (question 14).

4.4. Sharing in AFS

The existence of cross-cell file access in AFS is borne out by the data presented in Figure 5(b). That figure showed that the percentage of references to the files in foreign cells was up to 5% for data and up to 4.5% for status information during the 12-week data collection period. Although 5% may not seem like much, it is significant because cells represent organizational boundaries and most users tend to access data within their own organizations.

Table 8 represents a histogram of the number of different cells contacted by each client during the 12-week period. The table shows that two thirds of the clients referenced data in at least one foreign cell while 3% of the clients referenced data in all available cells. Further, examination of the raw data shows that, on average, 15% of the clients referenced foreign data each day.

We also repeated the study originally reported by Kistler and Satyanarayanan [4] on the extent of sequential write sharing on directories and files. Every time a user modified an AFS directory or file, the user's identity was compared to that of the user who made the previous modification. Our data, showing that 99.1% of

15. *What is the nature of AFS interaction between yourself and others in the same AFS cell? (Check any that apply.)*

- 3% No interaction
- 39% "Looking around" your cell's file space to see what's new
- 69% Reading files
- 15% Accessing AFS-based bboards
- 45% Copying interesting files into your own storage area(s)
- 33% Copying files in one direction (e.g., drop-offs)
- 39% Copying files back and forth, modifying them at each step
- 66% In-place use, modifying files without copying them
- 6% Other

16. *Have you ever explored/used the resources available through the grand.central.org cell?*

- 34% Yes, I've used the materials there
- 26% Yes, I've looked through it to see what's there
- 21% No, I haven't had the need/desire
- 19% No, the /afs/grand.central.org directory has not been set up at my site

17. *Have you ever explored/used the resources available at other cells?*

- 71% Yes, I've used the materials at other cells
- 20% Yes, I've looked through other cells to see what's there
- 8% No, I haven't had the need/desire
- 1% No, the /afs/<other cell> directories have not been set up at my site

18. *How many accounts/authentication identities do you have in other cells (i.e., how many cells other than your home cell can you klog to)?*

- 20% 0
- 36% 1
- 19% 2
- 18% 3
- 7% More than 3

19. *Rate the importance of the following communication/collaboration media and methods in your organization, using the following scale: 5: Very important, 4: Important, 3: Somewhat important, 2: Not important, 1: Not used at all.*

- 3.92 Direct phone calls
- 2.43 Conference calls
- 2.55 Internal (paper) memoranda
- 2.14 U.S. mail
- 2.69 Express/overnight delivery services
- 3.26 FAX
- 2.66 Physical media (floppies, hard disks, mag tapes, etc.)
- 4.56 Email
- 3.49 BBoards/Email lists
- 3.55 FTP
- 3.06 Local (non-distributed) file system
- 3.38 Non-AFS distributed file system (e.g., NFS)
- 4.05 AFS
- 0.17 Other(s)

20. *Are you currently working with people in another AFS cell on joint projects of any kind?*

- 7% Yes, frequently
- 8% Yes, moderately
- 23% Yes, but not very frequently
- 15% No, the people I collaborate with outside my own cell do not (all) run AFS
- 43% No (for any other reason)

Figure 8: Users' Perception of Sharing in AFS

all directory modifications were by the previous writer, is consistent with Kistler and Satyanarayanan's observations. Unfortunately, we are not able to report on write sharing on files due to a bug in the statistics collection tools.

These observations confirm that the wide-area aspects of AFS are indeed valuable. Our anecdotal data, presented in Figure 8, corroborates this conclusion. Most users rate AFS very highly as a communication and collaboration tool. In their local cell, over 60% of the users tend to read or modify files that do not belong to them (question 15). Most users have used or looked at materials that reside in other cells. About 80% possess accounts/authentication identities in foreign cells. About 38% of the users participate in joint projects with people from different cells, although 23% do not do so frequently.

Further anecdotal information of the value of wide-area file access is provided by highly visible instances of information dissemination and collaboration in AFS. For example, AFS has facilitated the development of OSF's DCE. It has also been used in the STARS project initiated in 1990 by ARPA, which established a nationwide government-commercial collaboration. In both these cases, wide-area file access has been used by participant organizations to support sharing and dissemination. Project software and documentation are located in AFS and collaboration via AFS has occurred on a regular basis. AFS has also been used as a tool for information dissemination. The release of MIT's X11R5 software is a good example. In September

1991, the X11R5 release was installed into the cell `grand.central.org` and all AFS sites were able to immediately browse and access the release without manual file transfers.

5. Conclusion

Our goals in conducting this study were to observe a wadfs in actual use and to characterize its usage profile. We were also interested in determining how well AFS worked at the current scale of the system, and to see if any imminent limits to its further growth were apparent.

The qualitative and quantitative data that we have presented confirms that AFS provides robust and efficient distributed file access in its present world-wide configuration. The caching mechanism is able to satisfy most of the file references from the clients' local cache. Even though file server and network outages can be disruptive for particular users, our observations show that prolonged server inaccessibility is rare. Our data shows no obvious bottlenecks that might preclude further growth of the system.

AFS's divide and conquer technique of using semi-autonomous cells for spanning widely disparate organizations has proven to be invaluable. By providing considerable flexibility in security and storage management policies, the cell mechanism reduces the psychological barrier to entry of new organizations. As a consequence, growth in AFS over time has not just been in the number of nodes in each cell, but also in the total number of cells.

In summary, this paper provides conclusive evidence that AFS is a viable design point in the space of wide-area distributed file system designs. We are convinced that any alternative design must preserve the aggressive caching policies and support for autonomous administration that are the hallmarks of AFS' approach. The absence of either of these features will be fatal in any attempt to build a file system that uses a wide-area network and spans many organizations.

Acknowledgments

The *xstat* data collection facility was designed and implemented by Ed Zayas. Contributions to the evaluation methodology for wide-area distributed file systems were made by Ed Zayas, Alfred Spector and Bob Sidebotham. Anne Jane Gray provided assistance in organizing this project. Comments by Mike Kazar, Maria Ebling, Qi Lu, and Jay Kistler were helpful in improving the presentation.

References

- [1] Baker, M.G., Hartman, J.H., Kupfer, M.D., Shirriff, K.W., Ousterhout, J.K., *Measurements of a Distributed File System*. Proceedings of the Thirteenth ACM Symposium on Operating System Principles, Pacific Grove, CA, October 1991.
- [2] Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N., West, M.J., *Scale and Performance in a Distributed File System*. ACM Trans. on Computer Systems, Vol. 6, No. 1, February 1988.
- [3] Kazar, M.L., *Synchronization and Caching Issues in the Andrew File System*. Usenix Conference Proceedings, Winter 1988.
- [4] Kistler, J., Satyanarayanan, M., *Disconnected Operation in the Coda File System*. ACM Trans. on Computer Systems, Vol. 10, No. 1, February 1992.
- [5] Morris, J. H., Satyanarayanan, M., Conner, M.H., Howard, J.H., Rosenthal, D.S. and Smith, F.D. *Andrew: A Distributed Personal Computing Environment*. Communications of the ACM, Vol. 29, No. 3, March 1986.
- [6] Ousterhout, J., Da Costa, H., Harrison, D., Kunze, J., Kupfer, M., Thompson, J. *A Trace-Driven Analysis of the 4.2BSD File System*. Proceedings of the 10th ACM Symposium on Operating System Principles, December, 1985.

- [7] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., Lyon, B., *Design and Implementation of the Sun Network Filesystem*. Summer Usenix Conference Proceedings, 1985.
- [8] Satyanarayanan, M., *A Study of File Sizes and Functional Lifetimes*. Proceedings of the 8th ACM Symposium on Operating System Principles, Asilomar, December 1981.
- [9] Satyanarayanan, M., Howard, J.H., Nichols, D.N., Sidebotham, R.N., Spector, A.Z. and West, M.J., *The ITC Distributed File System: Principles and Design*. Proc. 10th ACM Symposium on Operating System Principles, December 1985.
- [10] Satyanarayanan, M., *Integrating Security in a Large Distributed System*. ACM Transactions on Computer Systems, Vol. 7, No. 3, August 1989.
- [11] Satyanarayanan, M., *Scalable, Secure, and Highly Available Distributed File Access*. IEEE Computer, Vol. 23, N. 5, May 1990.
- [12] Satyanarayanan, M., *The Influence of Scale on Distributed File System Design*. IEEE Transactions on Software Engineering, Vol. 18, No. 1, January 1992.
- [13] Sidebotham, R.N., *Volumes: The Andrew File System Data Structuring Primitive*. European Unix User Group Conference Proceedings, August 1986.
- [14] Spector, A.Z., *Thoughts on Large Distributed File Systems*. Proc. of the German National Computer Conference, October 1986.
- [15] Spector, A.Z., Kazar, M.L., *Wide Area File Service and the AFS Experimental System*. Unix Review, Vol. 7, No. 3, March 1989.
- [16] Steiner, J.G., Neuman, C., Schiller, J.I., *Kerberos: An Authentication Service for Open Network Systems*. Usenix Conference Proceedings, Winter 1988.
- [17] Transarc Corporation, *AFS 3.1 Programmer's Reference Manual*. FS-00-D180, Pittsburgh, PA, October 1991.

Mirjana Spasojevic received the B.S. degree in mathematics from the University of Belgrade in 1986, and the M.S. and Ph.D. degrees in computer science from The Pennsylvania State University, in 1989 and 1991, respectively. She is currently working as a System Designer at Transarc Corporation. Prior to joining Transarc, she was an Assistant Professor at the School of Electrical Engineering and Computer Science, Washington State University. Her research interests include distributed operating systems and data management.

Mahadev Satyanarayanan is an Associate Professor of Computer Science at Carnegie Mellon University. He is currently investigating the connectivity and resource constraints of mobile computing in the context of the Coda File System. Prior to his work on Coda, he was a principal architect and implementor of the Andrew File System. Satyanarayanan received the PhD in Computer Science from Carnegie Mellon University in 1983, after a Bachelor's degree in Electrical Engineering and a Master's degree in Computer Science from the Indian Institute of Technology, Madras. He is a member of the ACM, IEEE, Sigma Xi, and Usenix, and has been a consultant to industry and government.

Wux: Unix Tools under Windows

Diomidis Spinellis
Department of Computing
Imperial College, London

Abstract

Wux is a port of Unix tools to the Microsoft Windows environment. It is based on a library providing a Unix-compatible set of system calls on top of Windows. Unix-derived tools run in parallel, communicating using the Unix pipe abstraction. All processes are run within an application template that gives them basic Windows compatibility such as input and output windows and an icon. The performance of the system is comparable to that of Unix ports to the PC architecture.

1 Motivation

The Unix operating system offers a wide variety of tools that can be used as building blocks or autonomous units for application development. Although these and similar tools can be ported to other environments [KP76, Gor93], their usability is often impaired by the lack of the glue elements available under Unix: multitasking and interprocess communication using pipes. Specifically, tool ports to the widely used MS-DOS system have to rely on serialised process execution and pipes implemented using inefficient intermediate files. The advent of the Microsoft Windows¹ system provided an accessible and widely used platform on which a more serious, complete, and efficient porting effort could be based.

The main objectives of our porting project were the following:

- Port a number of useful Unix-derived tools to the widely used Windows platform. As Windows applications have to be structured around a message loop, this exercise is far from trivial.
- Utilise the features provided by the Windows system to provide an environment that would be closer to Unix. Windows runs (in its 3.1 incarnation) on top of MS-DOS offering a number of advantages. The features important for our work were:
 - non-preemptive (co-operative) multitasking,
 - dynamically linked libraries with a data segment shared between processes,
 - memory management including virtual memory, and
 - a graphical user interface (GUI).

We hoped that the first three capabilities could be used to provide Unix-like functionality, while the last one would enhance the usability of the resulting system.

- Experiment with the integration of these utilities in the Windows environment. We wanted to test the possible synergy between the traditional Unix tools and Windows concepts like the mandatory message loop structure, the clipboard, and OLE (object linking and embedding).

2 Functional Description

Wux consists of a set of native Windows executable files, each one of them corresponding to a Unix tool (Figure 1). The programs can be used in two different ways:

¹From here on "Windows" is used to refer to Microsoft Windows.

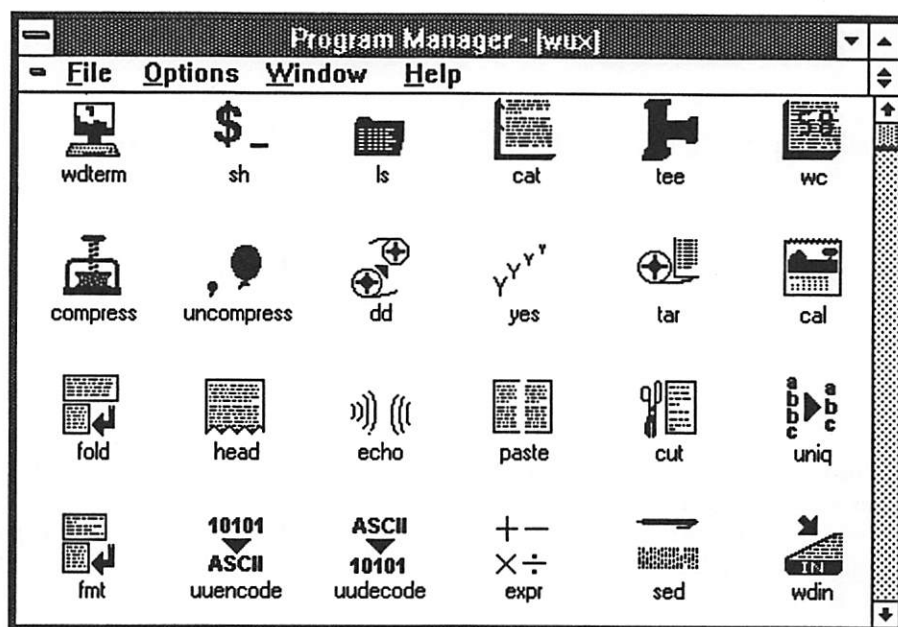


Figure 1: Program manager group of *wux* applications.

1. they can be executed from a shell-like window using the traditional Unix argument parsing and redirection functionality including pipes, or
2. they can be executed as stand-alone processes within the Windows environment using the Unix argument parsing and redirection conventions including redirection to file descriptors. To this second mode of execution we plan to add a graphical interface for argument specification and input/output file specification.

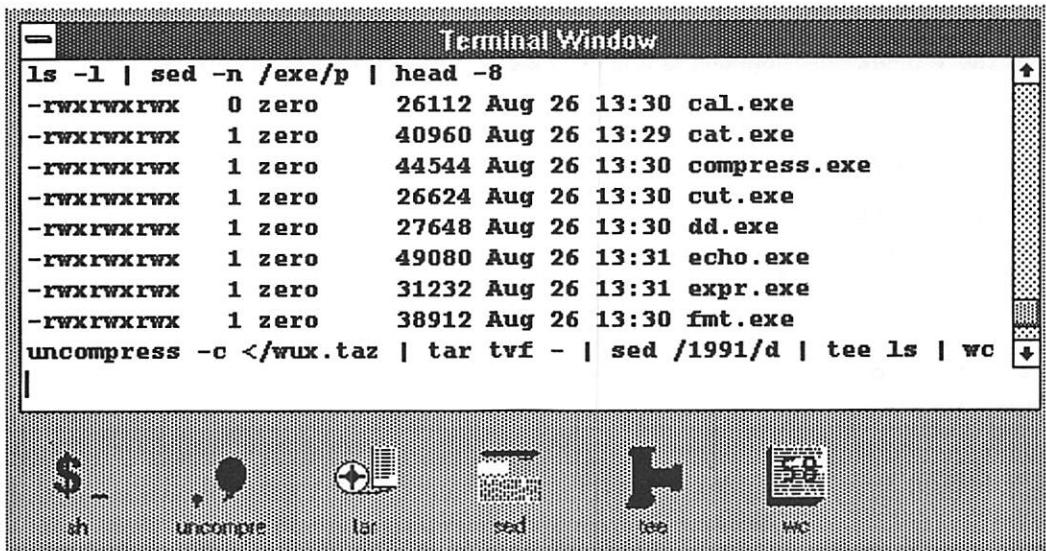
The programs run as native Windows processes, and can therefore utilise all the facilities provided by Windows, such as the ability to allocate extended and virtual memory, inter-process communication, and non-preemptive multitasking. Running processes are represented by their icon displayed at the bottom of the Windows screen (Figure 2). Whenever a process requests input from the standard input device or produces output to the standard output device and no redirection has been specified a suitable data source or sink window is created to provide that service. Data can be typed-in on a line-by-line basis in the *wdin* window which also provides an editable history of previous lines. Similarly, generated data can be displayed in the *wdout* window which provides a scrollbar to examine previous output. The standard error output is also redirected in a *wdout* window. Applications that are of interactive nature (such as editors and shells) can be executed with both input and output connected to *wdterm*, a terminal emulation window, akin to the X-Windows [SG86] *xterm* application.

3 *Wux* Implementation

In the following sections we describe the *wux* implementation. We first provide a short technical description of the Windows environment, then analyse the difficulties of porting Unix applications to Windows, proceed with an overview of the *wux* architecture and its components, and end with a section detailing how Unix applications are ported to *wux*, and some performance figures.

3.1 Technical Description of the Windows Environment

Windows 3.1 runs on top of the Microsoft MS-DOS operating system. In addition to a windowing graphical user interface, it provides many functions (see section 1) that are normally provided by an operating



```
Terminal Window
ls -l | sed -n /exe/p | head -8
-rwxrwxrwx 0 zero 26112 Aug 26 13:30 cal.exe
-rwxrwxrwx 1 zero 40960 Aug 26 13:29 cat.exe
-rwxrwxrwx 1 zero 44544 Aug 26 13:30 compress.exe
-rwxrwxrwx 1 zero 26624 Aug 26 13:30 cut.exe
-rwxrwxrwx 1 zero 27648 Aug 26 13:30 dd.exe
-rwxrwxrwx 1 zero 49080 Aug 26 13:31 echo.exe
-rwxrwxrwx 1 zero 31232 Aug 26 13:31 expr.exe
-rwxrwxrwx 1 zero 38912 Aug 26 13:30 fmt.exe
uncompress -c </wux.taz | tar tvf - | sed /1991/d | tee ls | wc
|
```

Figure 2: Windows screen running wux.

system. In contrast to the MS-DOS operating system it runs in protected CPU mode, and can thus isolate processes from disruptive interference and provide paged virtual memory.

All applications running under Windows must be structured around an event message loop. That loop typically retrieves messages from the per-process event queue and dispatches them to registered callback procedures that are part of each application. Messages can either be generated by the user (e.g. menu commands, mouse movement) or by the system (e.g. a window redraw request). During calls to the message retrieval functions Windows can pass control to other applications; this is how co-operative multitasking is implemented².

C programs can be compiled using the MS-DOS Intel x86 family compilers, but must be linked with a Windows-compatible library which provides a different startup code which calls *WinMain* instead of *main*. In addition, after the linking phase, a "resource compiler" attaches to the executable image of the program entities such as icons, string tables, menus, and dialog boxes. These can therefore be modified without re-compilation or relinking. Windows provides a facility of dynamic link libraries (DLLs). DLLs are shared libraries, that are loaded and linked whenever a function that resides in them is first called. Their code is executed in the context of the calling application, but their data segment is shared between all processes.

3.2 Difficulties in Porting Unix Applications to Windows

Many are the difficulties in porting Unix applications to the Windows environment. We can divide them into:

- the difficulties which arise from the requirements imposed upon conforming GUI Windows applications (startup sequence, need for an event loop),
- the differences between the Unix operating system and the "Windows over MS-DOS environment", and
- machine portability problems.

²Windows runs a number of virtual machines using a preemptive scheduler. These are however used to implement the MS-DOS compatibility boxes; only one of them (the system VM) runs the Windows GUI.

In the following paragraphs we provide an overview each of the three problem areas. In our discussion we assume that the Windows development is done using one of the standard software development kits and C compilers.

3.2.1 Windows-Specific Requirements

Programs that run as native Windows applications must provide a graphical user interface. At the very minimum, this should consist of an icon representing the program and a system menu that allows for the program's termination. For these to be provided, the program on startup must register a class and an event handling procedure. It must then continually retrieve and dispatch messages from the message queue in order to provide a functioning user interface and assure that other applications will be scheduled. On termination all resources allocated must be freed. It is very difficult to adapt existing non-GUI programs to this centralised control structure.

In addition, interactive character-based applications can not be directly executed, as Windows only provides a graphical front-end to native applications. The MS-DOS window application that provides a character screen, restricts programs to MS-DOS only functionality, and to a coarse grained and inefficient scheduler.

3.2.2 Operating System Differences

We will examine differences between the Unix and the "Windows under MS-DOS" environments from the perspective of the requirements for Unix compatibility. For this discussion we have used as a rough guide the POSIX.1 standard [ISO90]. Many of the programs that run on the Unix environment do not conform to this standard, and therefore our examination contains many additional pragmatic and practical problems, that would not exist in an ideal world. A thorough treatment of C program portability can be found in [Hor90]. The problematic areas can be divided among those related to the environment in which the applications are compiled, the environment in which processes execute, and the application programming interface.

Almost all Unix applications assume that they are to be compiled under the Unix environment; Windows is the platform of choice for compiling Windows applications. We will therefore combine our examination of the portability problems that arise from the differences in the compilation and execution environments. The problems related to the environment in which the processes execute are the following:

Missing or different executable files A number of Unix programs invoke other programs. Many programs rely on the Bourne shell [Bou86] to execute other programs with I/O redirection, or to perform wild-card expansion, others use one of the paging programs such as *more* or *pg* to pipe their output. Other examples are *rcs* which can execute *mail*, *crypt* which gratuitously executes *makekey*, and enhanced versions of *tar* which pipe through *dd*, *compress*, and *rsh*.

During the compilation phase applications sometimes rely on other tools such as *yacc*, *lex* or simpler text processing tools. Even when these tools exist under Windows they may be incompatible in the options they take, or the output they generate, or have lower processing size limits. Unix programs sometimes make assumptions about the C compiler used and contain *#pragma* commands, use non-portable pre-processor tricks, or expect certain variables to be placed in specific registers.

File names Many programs have source files, create temporary files, or need data files whose names are illegal for the MS-DOS file system (which is the one used by Windows 3.1). The problems can be filename size (only 11 characters are allowed), the use of one of the illegal characters " `, = + < > ; ? : ,`", the differences in case sensitivity, or the use of MS-DOS device names.

File system semantics A common convention for Unix programs is to delete temporary files after opening them, so that they will be automatically removed from the file system when the file is closed. This trick will destroy the integrity of the MS-DOS file system.

Missing or different files, directories and devices A number of files and directories that are standard on Unix systems (such as */etc/passwd*, */tmp*, and */dev/null*) either do not exist or are different under

MS-DOS. Unix programs that try to read executable, object, or library files will not run, as these formats are different under Windows.

Format of text files Lines in text files of Unix systems are delimited by a single line-feed (LF) character. The equivalent MS-DOS convention is a carriage-return (CR) followed by an LF. Some C libraries map CR/LF pairs to LF, but require the specification of the file type (binary or text) when the file is opened.

Environment variables Common Unix environment variables (*SHELL*, *HOME*) are not guaranteed to be defined under MS-DOS.

Command line size limits The program command line size limit under MS-DOS is only 127 characters. This can severely limit many applications, including the possibility of filename expansion and backquote substitution by the shell.

The third area of operating system differences encompasses the interface to the operating system. Since the code libraries are often part of the operating system in the following list we will also include library differences. The areas where the application programming interface using a Windows software development kit differs significantly from the environment found under Unix are the following:

Process creation, management, and termination There are no *fork* and *wait* system calls; the *exit* function can not be used.

Process environment There are no notions of user and group ids; the relevant system calls are therefore missing.

Exception conditions and handling Signal handling is different from the POSIX specification. Berkeley enhancements are also missing. Some signals commonly defined in Unix systems do not appear in *signal.h*.

File and Directory operations Some file status bits are not defined. *Pipe*, *select*, *link*, *symlink*, and *ftruncate/chsize* are missing.

File protection mechanisms The *umask* is not defined.

Record and file locking mechanism File locking uses a different set of operations.

Device specific functions There are no terminal controls, and I/O processing modes under Windows. The *fcntl* interface is completely different, and all constants and structures used by Unix programs are not available under Windows.

User and group database information These databases are not implemented under Windows and no functions are provided to access them.

Networking Although there is a sockets compatible API specification for Windows [HTA⁺93], implementations supporting it are not part of the standard Windows distribution.

Accounting No resource accounting is available under Windows and therefore no system and library calls are provided to access the relevant information.

System management As might be expected system management under Windows is totally different from Unix and therefore the relevant system calls (e.g. *reboot*) are not provided.

C Library differences The Unix C libraries are richer than the libraries normally available for development under Windows. Regular expression handling, directory access, screen handling (*curses*), and multiple precision arithmetic functions are not provided. The C locale specific functions are not integrated with the locale facilities available under Windows.

Many of the problems described above should cease to be issues when developing for the Windows-NT operating system which offers POSIX compatibility.

3.2.3 Machine Portability Problems

The third area of problems that arise in trying to port Unix applications to the Windows environment are general machine portability problems. Some are triggered by the “memory model” chosen for compiling the application. The memory model describes the scheme used to get around the addressing problems caused by the 16 bit address segmented architecture of the earlier Intel processors (from the 8088 up to and including the 80286). Different memory models offer different tradeoffs between code or data maximum size, and time or space efficiency. Since the choice of a memory model determines the properties of C pointers, some memory models reflect better the assumptions made by some Unix programs than others. The problems most commonly exhibited are:

Integer size Windows development is still mostly done using 16 bit compilers in order to maintain compatibility with 80286 based platforms. Many Unix programs are coded with the assumption that the integer size is 32 bits and misbehave when compiled for Windows. Under certain memory models the size of pointers is 32 bits. Some programs assume that pointers and integers are of the same size, and therefore fail to run correctly.

Memory size Some memory models allow only 64K of data or code. Many Unix programs assume that a large (often virtual) memory pool is available and terminate with a memory allocation error. Furthermore, unless a severe drop in efficiency can be tolerated, it is not possible to create individual data objects (e.g. a structure, or an array) larger than 64K. Functions whose code would be larger than 64K can not be compiled.

Machine dependencies Some older Unix programs depend heavily on a particular machine, containing VAX assembly instructions, or native code generators.

The integer and memory size problems will be solved once development shifts to the newer 32 bit compilers that produce code for the 80386 and newer processors.

3.3 *Wux* Architecture

The *wux* Unix tools run on top of a set of support libraries that act as intermediaries between the tools and the Windows kernel (Figure 3). The support libraries consist of a statically linked library that provides Unix compatibility replacing or adding a number of functions to the standard C library, and a DLL that provides the file descriptor interface including support for the *pipe* system call. In addition, a standard application template is provided that is linked with all *wux* applications to provide the initialisation, message processing, and termination functions required by the Windows environment.

The *wdin*, *wdout* and *wdterm* windows described in section 2 are implemented as separate Windows applications that respectively generate output lines from the user interaction, display input lines on the screen, or combine the two functions.

3.4 The File Descriptor Compatibility Library

The *wux* file descriptor library is layered on top of the compiler C library to provide functionality similar to that of the Unix system on top of Windows. All calls to the C library are intercepted and handled by the *wux* library. The function substitution is implemented by a set of *#defines* in a header file which is included by all other header files. Table 1 contains the Unix system calls handled by *wux*, while table 2 contains the new function calls that have been introduced. *Wux* maintains in shared memory an internal table of file descriptors which contains the information needed to implement pipes. Calls referring to real files are passed down to the C library, while calls that refer to pipes are handled internally. Inter-process communication is handled using shared global memory. Pipes are implemented in a manner analogous to the implementation described in [Bac86, pp. 113-116]. Processes sleeping on empty or full pipes loop in a Windows *PeekMessage/GetMessage* loop in order to handle process messages and allow the Windows non-preemptive multitasking to execute other processes. We decided to program pipes at a low level instead of relying on the IPC mechanisms provided by Windows in order to avoid unwanted interactions between the Windows user interface messages and the IPC messages.

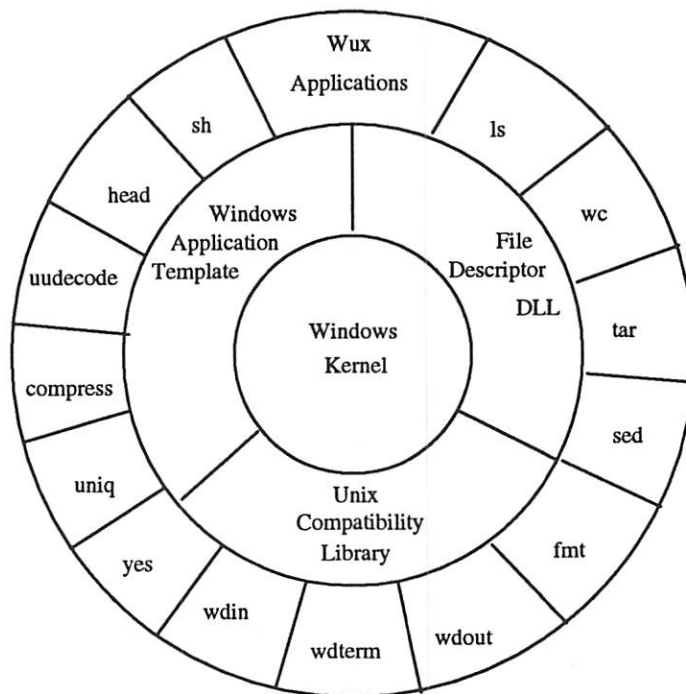


Figure 3: *Wux* structure.

File descriptors are indexes to a fixed sized array of structures containing information for every file descriptor. The structure contains bits that indicate whether a particular file descriptor is used, closed, refers to a pipe, and its read/write mode. In addition, every structure contains a reference count which is used to mark file descriptors as unused. For file descriptors that refer to operating system files, the structure contains a handle to that file, whereas for file descriptors that refer to pipes the structure contains a pointer to a buffer residing in global (shared) memory, an offset into that (circular) buffer, the number of bytes residing in the buffer, and a *longjmp* environment used for termination message handling (discussed in section 3.7).

3.5 The Unix Compatibility Library

The Unix compatibility library provides a number of functions found in the Unix C library, but not available under MS-DOS, such as regular expression matching and directory access, together with the Berke-

```
int wux_open(char *fname, int flags, int mode);
int wux_creat(char *fname, int mode);
int wux_pipe(int fd[]);
int wux_read(int fd, void *buffer, int count);
int wux_write(int fd, void *buffer, int count);
int wux_readv(int d, struct iovec *iov, int iovcnt);
int wux_writev(int d, struct iovec *iov, int iovcnt);
int wux_close(int fd);
int wux_lseek(int fd, long offset, int whence);
int wux_fstat(int fd, void *buf);
int wux_isatty(int fd);
int wux_getdtablesize(void);
```

Table 1: Unix compatible *wux* functions

Function	Description
<code>int wux_fdref(int fd);</code>	Increment the file descriptor reference counter.
<code>int wux_clmark(int fd);</code>	Mark file descriptor to start up a <i>wdin/wdout</i> process if used for I/O (lazy process startup).
<code>int wux_fileno(int fd);</code>	Return the underlying MS-DOS file descriptor (if <i>fd</i> does not refer to a pipe).
<code>int wux_setexitenv(int fd, jmp_buf env);</code>	Associate a file descriptor with a <i>longjmp</i> stack environment for process termination.

Table 2: New *wux* functions.

ley 4.4 BSD *stdio* library compiled to call the *wux* functions instead of the C library functions. We had to provide our own version of the *stdio* library, because the *wux* function substitution (e.g. calls to *read* mapped to *wux_read*) is performed at compile time.

3.6 The Windows Application Template

The Windows application template provides a standard wrapper for implementing the Unix tools. It is a Windows program that handles window creation, argument parsing and I/O redirection, and creates a process exit environment to be used by the *exit* function and the window kill message handling code. It then invokes the *main* function of the tool that was linked to it. Input/output redirection has to be handled in the context of the process using the files, as file descriptors are not inherited under Windows.

3.7 Control handling in *wux* applications

Handling of control in *wux* applications is complicated, as the normal control flow of the Unix tool must be modified to accommodate Windows message handling. In addition, the C *exit* function is not compatible with the Windows termination sequence, because Windows programs are expected to terminate by destroying the application window and exit by returning from the Windows entry point procedure — which normally contains the application event loop. Figure 4 illustrates the control flow between Windows, the *wux* support libraries, and a *wux* application. On application startup, Windows calls the application entry procedure *WinMain*, which in our case resides in the application template. That procedure performs the startup processing described on section 3.6, and then calls the C *main* function to start the execution of the tool proper. Whenever the tool calls a *read* or *write* function, and these have to wait (due to an empty or full pipe buffer), a *sleep* function is called which loops around a Windows message loop getting messages from the application queue and dispatching them to the appropriate handler function (*WndProc*). It is inside that loop that Windows gets a chance to reschedule other applications as runnable. The handler function in our case also resides in the application template code, and does nothing more than check for application termination, and pass control back to the default Windows handler function (*DefWindowProc*) which does the proper things for most messages (window dragging, iconisation etc.). If the message loop detects a *WM_QUIT* message (which can be generated when the user destroys the application's window, or selects *terminate* from the application's system menu) it terminates the application by transferring control using *longjmp* to the end of the application template main procedure which returns to Windows. *Longjmp* with the same target address is also called whenever the *exit* function is invoked. The two cases are however dissimilar, as the first one corresponds under Unix to the termination of a program by a signal whereas the other corresponds to a normal program exit.

3.8 Implementation of the shell and the *wdterm* applications

The shell and *wdterm* applications are special, because they interact with the Windows environment, and are based on the *wux* implementation.

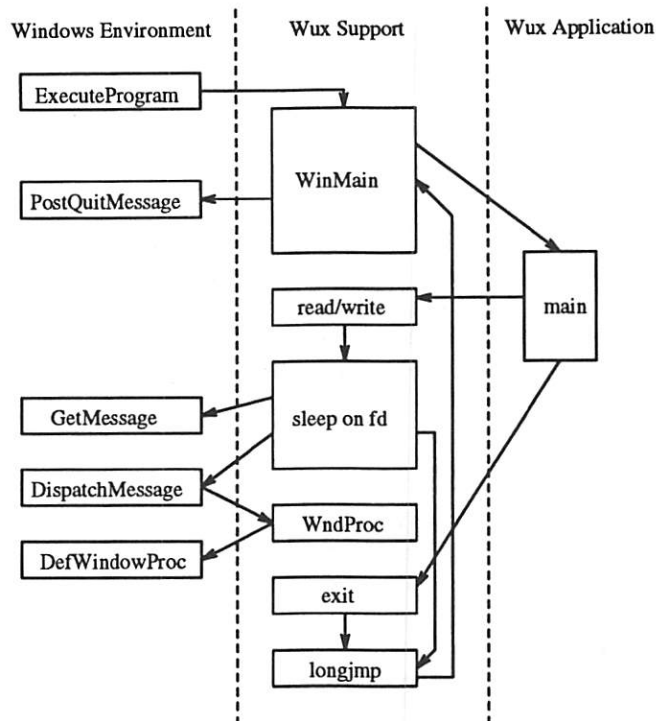


Figure 4: Control flow between Windows, the *wux* support libraries, and a *wux* application.

Wdterm is a Windows application that displays its standard input in a window, and sends user text input on its standard output. When its standard input and output are connected to a shell it forms an interface similar to that of a terminal running that shell. It is implemented using a Windows code template similar to that used for all other *wux* tools, but modified to contain two additional elements:

- a scrollable list region which is used to display the user input and command output, and
- an edit line which is used to enter new commands or text.

A single function (*addline*) is used to append lines into the scrollable region. When the region grows to a certain size, every append also deletes the topmost line in order to limit the amount of memory required. The message processing function of the application is modified to provide the scrollbar functionality, the ability to re-execute commands from the scrollable region by double-clicking them, and a handler for the “return key pressed in the editable region” message. That handler just gets the user input from the editable region, appends it to the scrollable region and prints it on standard output:

```

case WM_COMMAND:
    switch (wParam) {
        case IDOK: /* Return key */
            (void)SendMessage(from, WM_GETTEXT, textlen + 1, (LPARAM)(LPCSTR)txt);
            addline(to, txt);
            fputs(txt, stdout);
    }
  
```

The handling of input to *wdterm* is handled in an even simpler way. After the window is created, the program enters a loop that reads lines from the standard input and adds them to the scrollable region:


```
while (fgets(buff, sizeof(buff), stdin))
    addline(hWndList, buff);
```

The *wux* message handling assures that messages (such as user input, and scrollbar manipulation) are correctly handled while that loop is executing.

The shell is implemented like all other Unix tools, but uses Windows calls and *wux* conventions to handle command execution and the setup of pipes. In the future we plan to isolate the *wux* dependencies and port one of the publicly available Unix shells to *wux*. All commands are executed asynchronously, because we have not yet implemented a *wait* function call. As file descriptors are not inherited under Windows, input/output redirection, and pipe setup can not be handled by the shell. For this reason pipes are implemented by passing the file descriptor number of the appropriate pipe end to the application's command line using the ">&n" syntax.

3.9 Porting Applications to Wux

Porting of Unix applications to the *wux* environment is relatively straightforward. The *wux* include files and libraries contain a number of definitions and functions that minimise the porting effort. In some cases absolutely no code modifications are needed. The code is simply recompiled using a standardised Makefile and definition file (a file needed by the Windows linker). A suitable icon should be created before compiling. If the application to be compiled comes with a non-trivial makefile, then the application's makefile needs to be tailored after the *wux* makefile template. Possible compatibility problems can appear:

- at compile time due to missing tools, include files, or undefined constants,
- at link time as unresolved references to unimplemented functions, and
- at run time if some aspect of the Windows environment does not match the application's expectations.

As we port more and more tools, we try to rectify these problems by integrating the solutions into the *wux* environment.

3.10 Performance

Although we expected the performance of *wux* to suffer from the layering approach and the Windows overhead, we were pleasantly surprised to find it comparable to other systems providing similar functionality. Table 3 details the times needed to copy data between two processes using a pipe. These include the context switch time. All measurements were made on the same machine and — with the exception of the MS-DOS case which used intermediate files — no disk activity. The speed of the Linux Unix system was expected since it is coded using the multitasking features of the 80386 processor family. The speed of the *wux* implementation can be explained by the fact that Windows provides only rudimentary process isolation and can therefore perform fast context switches. The slowness of MS-DOS is due to the intermediate files used to implement pipes. Although the tested systems are not directly comparable, the figures provide a rough guide on the relative performance of applications relying heavily on data transfer via pipes.

	Block size in bytes		
	1	2048	4096
Linux	220	325	375
Wux	295	988	1480
NetBSD 386	678	1945	3070
MS-DOS	492	67175	86250

Table 3: Time (in μs) needed to copy a chunk of data.

4 Related Work

The portability of tools similar in scope with those available under the Unix environment is examined in [KP76], who also detail how operating system deficiencies can be overcome. A port of the AT&T Unix tools for MS-DOS machines is commercially available in the form of the MKS toolkit [Gor93]. The offering is complete, well documented and integrated. It does not however provide multitasking and real pipes. A similar but non-commercial port of the project GNU Unix utilities for MS-DOS machines is available under the name "GNUish MSDOS" [OGH⁺93]. These two ports can run in the Windows environment only in the MS-DOS compatibility box, suffering all the restrictions of MS-DOS. User-level Unix emulation on top of another operating system (Mach) is described in [GDFR90], while [Fra93] provides a detailed analysis and specific solutions on the portability problems that can arise between dissimilar operating systems (Oberon and the Apple Macintosh). Finally, the G shell environment [MLS88] attempts to integrate the Unix tools with the X-Window system.

5 Further Work

Although we already use *wux* for day-to-day work, there are many possibilities to extend it making *wux* a lot more useful. We are currently working on porting more Unix applications to *wux*. As 32 bit compilers for Windows become more mature we hope to be able to port more substantial tools like Perl [WS90] and Jef Poskanzer's portable bitmap collection and integrate them with the Windows environment. Other possibilities we are exploring are a graphical user interface for file selection and command argument specification, the integration of the Unix manual pages into the Windows hypertext help system, and the integration of the Windows clipboard. Having laid a foundation for Unix tools and interfaces we would like to experiment with using the Unix tool-based approach to work with multimedia Windows data.

6 Conclusions

The Windows multitasking features allow the realisation of pipelines of concurrently executing programs. The globally shared memory provided by the Windows dynamic link libraries can be used as the in-core pipe buffer, and the Windows message loop as a synchronisation mechanism. A careful implementation of a compatibility system consisting of modified C headers, support libraries, and a Windows application template, makes it possible to port a number of useful Unix tools to Windows by recompiling the source code. The tools can be used as building blocks for the creation of a Unix-like working environment. We plan to extend the system providing better integration with the Windows features.

Acknowledgements

I would like to thank Stuart McRobert, Jan-Simon Pendry, Periklis Tsahageas, and the anonymous referees for their helpful comments on earlier drafts of this paper.

References

- [Bac86] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1986.
- [Bou86] S. R. Bourne. An introduction to the UNIX shell. In *UNIX Users' Supplementary Documents*. Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720, April 1986. 4.3 Berkeley Software Distribution.
- [Fra93] Michael Franz. Emulating an operating system on top of another. *Software: Practice & Experience*, 23(6):677-692, 1993.
- [GDFR90] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. Unix as an application program. In *Proceedings of the Summer 1990 Usenix Conference*, pages 87-95, Anaheim, USA, June 1990. Usenix Association.
- [Gor93] Ian E. Gorman. Building a portable programming environment. *Dr. Dobbs' Journal*, 18(5):76-81, May 1993.

- [Hor90] Mark R. Horton. *Portable C Software*. Prentice-Hall, 1990.
- [HTA⁺93] Martin Hall, Mark Towfiq, Geoff Arnold, David Treadwell, and Henry Sanders. *Windows Sockets: An Open Interface for Network Programming under Microsoft Windows*, version 1.1 edition, January 1993. Available via anonymous ftp from microdyne.com:/pub/winsock.
- [ISO90] International Organization for Standardization, Geneva, Switzerland. *Information technology — Portable operating system interface (POSIX) — Part 1: System application programming interface (API) (C Language)*, 1990. ISO/IEC 9945-1.
- [KP76] Brian W. Kernighan and P. J. Plauger. *Software Tools*. Addison-Wesley, 1976.
- [MLS88] Rick Macklem, Jim Linders, and Hugh Smith. G shell environment. In *Proceedings of the Summer 1988 Usenix Conference*, pages 15–22, San Francisco, USA, June 1988. Usenix Association.
- [OGH⁺93] Thorsten Ohl, Jean-loup Gailly, Ken Holmberg, Mark Lord, Russell Nelson, Len Reed, Stuart Phillips, Ian Stewartson, and other contributors. GNUish MSDOS. Available via anonymous ftp from wuarchive.wustl.edu:/systems/ibmpc/msdos/gnuish, January 1993.
- [SG86] R. W. Scheiffler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.
- [WS90] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly and Associates, Sebastopol, CA, USA, 1990.

Trademarks

UNIX, is a registered trademark of USL/Novell in the USA and some other countries.
 Microsoft and MS-DOS are registered trademarks, and Windows is a trademark of Microsoft Corporation.
 VAX is a trademark of Digital Equipment Corporation.
 80386 is a trademark of Intel Corporation.

Biography

Diomidis Spinellis is currently designing and implementing heterogeneous environment software tools for SENA S.A. in Athens, Greece. Diomidis has an M.Eng. degree in Software Engineering from Imperial College (University of London) and is really close to completing his Ph.D. on multiparadigm programming at the same institution. In the last ten years he has provided consulting services in the areas of CAD, product localisation, and multimedia. His research interests include software tools, operating systems, and programming languages. He can be reached via surface mail at SENA S.A., Kyprou 27, 152 37 Filothei, Greece, or electronically at dspin@leon.nr cps.ariadne-t.gr.

An MS-DOS File System for UNIX

*Alessandro Forin
Gerald R. Malan*

*School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213*

Abstract

We have written DosFs, a new file system for UNIX that uses MS-DOS data structures for permanent storage. DosFs can be used anywhere a traditional UNIX file system can be used, and it can mount disks written by MS-DOS as regular UNIX partitions. DosFs can be used as the root partition, and exported by an NFS server. Our motivation for this work was efficient disk space utilization; DosFs provides from 30% to 80% better disk utilization than the 4.3 BSD Fast File System (FFS). In addition, we found that the disk block allocation algorithm used by DosFs lets us exploit large contiguous disk operations, providing a five-fold improvement over FFS for uncached operations. To the best of our knowledge, this is the first file system implementation that allows the user to specify the logical block size at mount time. A user can mount the same filesystem with a larger block size when file accesses tend to be sequential and a smaller one when they tend to be scattered. The MS-DOS structures were designed for a single-user system and do not support access control. We solved this problem with simple extensions that are backward-compatible with MS-DOS.

1. Introduction

Today's micro-kernel architectures make it possible to run multiple operating system environments simultaneously. For example, a notebook computer running Mach 3.0 [2] can have both UNIX and MS-DOS disk partitions. In fact, an MS-DOS emulator [12] can be started from within the UNIX environment so that Windows applications can run in parallel with UNIX applications. Unfortunately, UNIX cannot access files on the MS-DOS partition while the MS-DOS emulator can access all UNIX files. The perception from UNIX is that part of the disk is missing, causing inconvenience both in resource utilization and in the sharing of data between applications that belong to the two different worlds. Other commercial MS-DOS emulators have similar limitations. A number of toolsets do exist to let UNIX users access MS-DOS floppies and disks, but they only transfer data between the UNIX and MS-DOS file systems. The best solution is an integrated file system that empowers all existing tools and applications.

We had both technical and practical motivations for this work. On the technical side, we wanted to examine and compare the permanent data structures used by MS-DOS and FFS, check whether it was possible to extend a single-user file system like the MS-DOS one into a multi-user safe, networked file system, and understand some initial performance figures for uncached disk accesses. On the practical side, we wanted to extend our UNIX system to support more disk formats, including MS-DOS and the ISO 9660 [5] disk formats. Some of our users expressed a desire to have the UNIX backup and restore utilities work on their MS-DOS data. Finally, a UNIX local filesystem should be usable as the root file system for a multi-user UNIX system.

There are very few file systems for UNIX that handle permanent storage, the latest BSD 4.4 release has added

LFS to the only one that was previously provided. There are a few other file systems outside of the BSD sources, but they all use permanent data structures similar to the original UFS ones. The permanent data structures used by MS-DOS are different enough from the standard UNIX ones to make it interesting to explore the effects on functionality and performance. For example, there are no inodes in MS-DOS.

The MS-DOS file system was designed for a single-user environment, on a personal computer. UNIX is used in multi-user, possibly security-sensitive environments. On security grounds, the single-user file system model is not acceptable on UNIX. For instance, MS-DOS has no provisions for storing user ids or access control information, but it is not acceptable to make files and directories readable and writeable by all users. The models of the expected file system usage might instead be closer. On UNIX, it is common to assume a time-sharing model, such as was traced by John Ousterhout [10] in his investigations of the 4.2 BSD file system. But even the most powerful workstations today are typically used by a single user at a time, and the file systems themselves use sophisticated caching strategies. It might well be that the MS-DOS model is quite appropriate for most UNIX users after all.

In our initial tests we found that native MS-DOS 5.0 was up to five times faster than our emulated UNIX (BSD 4.3 on top of Mach 3.0) for large uncached reads, and four times faster than the monolithic Mach 2.5. We traced the disk operations with a SCSI bus analyzer to find an explanation. We found that the MS-DOS file was contiguously allocated and was being accessed with four times larger block sizes (e.g. 16 KB at a time) just like our user program was requesting. The FFS file was allocated in an interleaved fashion to cope with rotational delays [8], and was being accessed at the file system block size, 4 KB. The extra seek times accounted for the remaining discrepancies.

Assuming we were able to reach the same optimal disk allocation for files as in MS-DOS, we realized that we could provide new functionality to our users, namely the ability to specify the disk access size (logical block size) dynamically, at mount time. Depending on prevailing use, with DosFs we can either use a small or a large block size, whichever gets the best performance, without any need for rebuilding the file system. A smaller block size is preferable in case buffer-cache misses are non-sequential and dispersed across the file system. A larger block size is best in the more common case that accesses tend to be sequential and clustered around a few files at a time.

Finally, we wanted to support the ISO file system organization (and the RockRidge extensions for POSIX/UNIX [13]) that is standard in CD-ROM disks. We started the project with some code written by Pace Willisson at Berkeley, which provided a read-only ISO file system implementation. The code was missing the RockRidge (RR) extensions and the file handle operations for use with NFS. Both were easy to add. Support for the MS-DOS file system looked, at first, like another simple addition because the ISO file system has various similarities with the MS-DOS file system. We just had to add the write-part--- which turned out to be more than twice the initial code.

The remainder of the paper is structured as follows. Section 2 describes the organization of the MS-DOS file system and permanent disk data. We assume the reader has some knowledge of the BSD FFS data structures, descriptions can be found in [8, 7]. Section 3 details the main design and implementation problems. Section 4 illustrates the programs we wrote for file system maintenance. In Section 5 we look at the performance optimizations from the following viewpoint. We use native, uncached MS-DOS as the base for each test. Each optimization is described and its individual effect quantified relative to either the base or the final performance figure. In many cases we can use run-time flags to toggle a specific optimization on or off to help us understand the measurements. Section 6 reports on the first evaluation criteria we used, namely the disk utilization profiles for FFS and MS-DOS. Section 7 describes the testing setup and the results of various performance tests. The tests themselves were taken from those commonly used in the file system literature and were selected to illustrate the effects of one or more performance optimizations. In Section 8 we point out similar and related work and finally draw a few conclusions from our experience and illustrate the status and availability of the code.

2. The MS-DOS File System Structure

Starting from the first sector of a disk, the MS-DOS file system layout is as follows; further details can be found in [1]. The initial section of the disk contains the primary (1 sector) and the secondary bootstraps (a variable number of sectors). This is followed by one or more copies of the File Allocation Table (FAT), which is followed by the root directory entries. The rest of the disk is divided into dynamically allocated *clusters*. Figure 2-1 graphically depicts this layout.

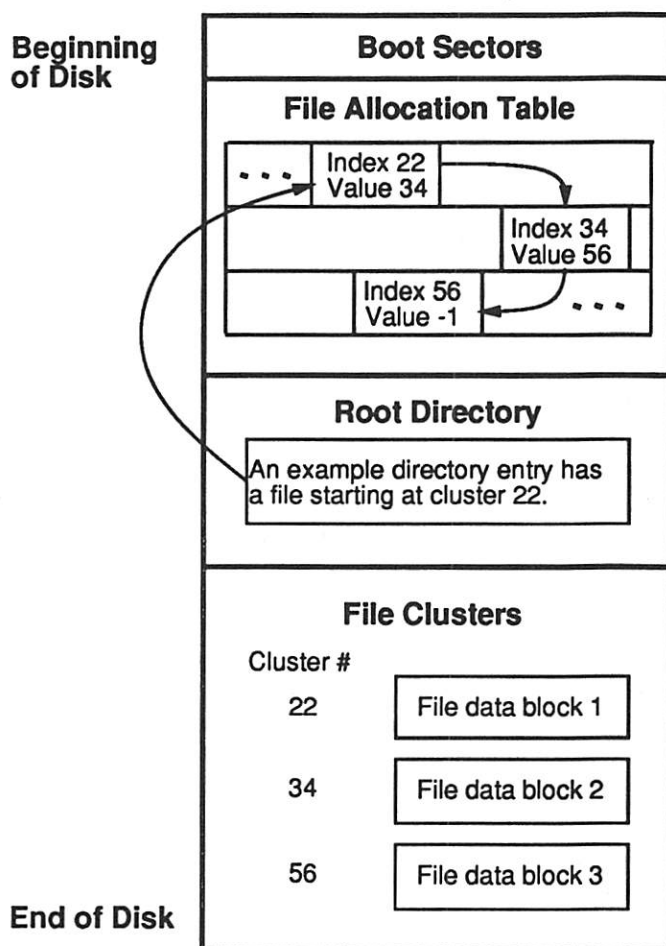


Figure 2-1: Organization Of The MS-DOS File System

The first sector of the disk contains the primary bootstrap code, intermixed with geometry information about the disk and possibly partitioning information. It also records the location, size and number of FATs, the size of the root directory, and the size of physical sectors and clusters. The primary bootstrap code uses this information to load the secondary bootstrap code into memory and transfer control to it.

The FAT is a fixed size table that describes the allocation status of the disk's clusters. Entries are either 12 or 16 bits wide, depending on the total number of clusters in the disk partition. A floppy has a 12 bit FAT, a fixed disk a 16 bit FAT. The first two FAT entries are reserved. MS-DOS uses a single FAT copy on floppy disks, and two copies on fixed disks to protect against disk corruption. DosFs accepts and keeps coherent any number of copies of the FAT.

To determine which disk blocks are allocated to a file (or directory) we must start from the file's directory entry. The entry contains the file's starting cluster number, e.g. the first block of the file. To find the next block, we use the cluster number as an index in the FAT. The value at that entry is the cluster number of the second block, and so on. The last block of the file has a FAT entry value of -1. If a cluster is not allocated, its FAT entry is zero. Other distinguished values are used to indicate reserved clusters (bad blocks, for instance).

Directory entries are fixed in size, 32 bytes each. Every entry contains the name and extension, an attribute byte that indicates, among other things, if this is a directory or a file, a timestamp, the size of the entry (zero for directories), and the starting cluster number. There is room for 10 more bytes that are defined as reserved by the file system specifications. The upper two bits of the attribute byte are also reserved/undefined. In the initial versions of MS-DOS, the root directory was the only directory and it could not grow. Later versions of MS-DOS are backward compatible and also have a fixed, contiguously preallocated root directory. The root directory is therefore special, fixed both in size and disk location at file system creation time. Other directories can be grown/shrunk as necessary. Some directory entries are special, and identified as such by the attribute byte. The MS-DOS image itself, for instance, is marked invisible and read-only. Label entries contain information about the creation time of the file system, and the volume name. There is only supposed to be one such label entry, in the root directory.

Clusters are logical blocks, e.g. multiples of the physical sectors. Their size is typically 512 bytes on small floppies and 2048 bytes on fixed disks. Just as in FFS, any cluster can contain either file data or directory information. Unlike FFS, no other type of information (metadata) is recorded in the clusters. The cluster size can actually be set to any power of two multiple of the physical sector size, similarly to the fragment size in the UNIX BSD file system. Unlike BSD, the cluster size can be up to a 16 bit multiple of the sector size. In practice, the upper limit supported by MS-DOS is 64 KB.

3. Design Issues

The MS-DOS file system evolved from its CP/M predecessor, it is aimed exclusively at a single-user machine, with no protection between users of the file storage. It strives to minimize resource utilization, both in memory and on disk. But it lacks features present in UNIX file systems, such as file links.

DosFs is implemented as a layer of code below the virtual file system (Vnode) layer [6]. The Vnode layer has a well defined interface, which must be fully implemented to cover all aspects of UNIX file system functionality. This presented us with various design issues, which we will describe in the rest of the section. Table 3-1 summarizes these issues, and indicates how they are solved in the various file systems.

<u>Problem</u>	<u>FFS</u>	<u>IsosFs</u>	<u>MS-DOS</u>	<u>DosFs</u>
Access Control	inodes	extended attributes	none	directory ext. or mount option
Filename size	255	30 or RR unlimited	11	11
Filename chars	ASCII	Uppcase+	Uppcase++	mount option
Inode numbering	Inode Table	Extent num	N/A	Cluster num
Links	hard+soft	hard+soft with RR extensions	none	soft
NFS access	inode num	inode + dir	N/A	inode + dir
Special devices	standard	RR ext.	no	directory ext.
Setuid files	standard	RR ext.	no	directory ext.

Table 3-1: Permanent Data Structures And UNIX Functionality Issues

3.1. Protection

In FFS, a directory entry contains only the name of the file and the inode number. A separate table, the inode table, contains a 128 byte record for each inode with various information, including the user-id, group-id and access permissions of the inode, its type and (a portion of) the list of blocks owned by the inode. There are no inodes in the MS-DOS file system, and a directory entry does not have the user-id, group-id, or access mode information (except for read-only files). There are two approaches for adding access protection to DosFs. We could use the 10 reserved bytes to store the extra information or we could dynamically calculate the access permission for each file.

The first approach is acceptable if MS-DOS applications ignore those 10 bytes (they do) or if backward compatibility and interoperability are not concerns. It is our goal to interact as smoothly as possible with MS-DOS, including potential applications that might themselves store information in those reserved bytes. Therefore we provide the extended information option only if DosFs is mounted as the root partition, or if explicitly requested via mount options. In practice, we have not yet found an MS-DOS application that would look at the content of reserved bits and bytes, nor does MS-DOS itself complain or reset these bytes. A disk utility (such as our *dosfsck*) can simply reset the extra protection information, to guarantee 100% compatibility with MS-DOS. We have designed our extensions so that no data is lost if the access information is destroyed.

For the second approach there are several possibilities:

1. [root,wheel] owns all files; write permissions depend on mount mode (read-only prevents all writes) and on the read-only bit in the attribute byte; all users have equal access rights;
2. same as above, but the user who issued the mount command is substituted for [root,wheel]; access modes are taken from the umask and propagated to all files; other users have access rights as per the umask;
3. allow the user to specify an arbitrary user-id, group-id, and access mode mask; otherwise, same as #2 above;

All three options essentially give the same ownership to all files, they just specify this ownership differently. We selected option #2 as default because it allows a user to mount his/her volumes and access them safely via NFS without any special options. Privileged users or operators can use option #3.

Regardless of the mount options, if a file on disk has the special extension marker we use the protection information stored on disk; only the mode mask is used to mask off specific bits. An interesting use of the user-specified mode mask is to disable setuid programs or execute permissions, regardless of their value on disk. This could be used by a cautious operator when mounting suspicious floppy disks on a general-purpose machine, to get rid of Viruses.

A possible alternative was to make use of the special *label* directory entries. MS-DOS specifies that they are optional and they would only appear in the root directory. We could put a label entry on each directory, to store the protection information. This idea is similar to the AFS Access Control Lists, which are set on a per-directory, not per-file basis [14]. We were told by Microsoft that it is not a good idea to misuse label entries.

The ISO file system defines *extended attributes* that provide protection information, including owner and group identifiers, and permissions. These are stored along with the file data in the first block of the file, separated from any directory entry information. ISO leaves open the mapping of this information onto the receiving system's administrative database.

3.2. File Naming

MS-DOS file names are limited to 12 characters, 8 bytes of name proper, an implied dot character to separate the file extension, and three characters of extension (suffix). The ASCII space character must be used to pad both the name and the extension fields. The MS-DOS specifications only dictate 8-bit ASCII characters but some internal MS-DOS functions convert all characters to uppercase. Therefore in practice all file names in MS-DOS have uppercase only ASCII characters, and digits. Special punctuation symbols are discouraged; some might produce peculiar effects.

The ISO file system sets a total limit of 30 characters on a file name, not counting the dot separator between name and extension and the trailing version information (a semicolon followed by a digit). The only permissible characters are digits, uppercase letters, and underscore. The RockRidge (POSIX) extensions use only the ASCII codes for letters and digits, period, underscore and hyphen. Extra information is added to a directory entry to express a file name in this POSIX format and bypass the ISO restrictions. This file name encoding scheme has no maximum length.

UNIX BSD file names have a set limit of 255 characters, and no restrictions on their content except for the first seven bits of the ASCII code. In principle, raising the 255 limit would be just a matter of modifying one define and recompiling. There is only one function that imposes the seven bit restriction in our 4.3 BSD file system code.

To cope with this jungle of rules and exceptions we wrote separate file name comparison and translation functions for the FFS, DosFs and the ISO file systems. In the case of the ISO file system, these handle the RR extension and recover the original UNIX names, bypassing the ISO limitations. The file name length limitation for MS-DOS instead remains, we are still looking for an acceptable workaround, such as using label entries or a separate string table.

We did remove the uppercase-only limitation: at mount time it is possible to specify one of three translation options. File names can be untranslated (except for the implied dot between name and extension) and look exactly as they would under MS-DOS. Or we can map letters to lowercase and clean up the extra spaces, to make file names look more UNIX-like. This is the default option. When used as a root file system we use a third scheme that treats name and extension as a single string, without any implied "dot" and without case conversions. This third option provides the most "proper" feeling for UNIX users, but it might make the file name unacceptable to native MS-DOS.

3.3. Inode Numbering

FFS maintains permanent inode tables on disk, and there is a direct mapping from inode number to disk address. Since ISO file systems do not have inode tables we used the first block number of directories and files (the *extent*, in ISO parlance) as the inode number. We maintained this scheme for the DosFs file system, the only modification is that our *dosfs_ialloc()* function must procure a cluster each time we create a new "inode". For directories this is not a problem since a directory by definition must contain at least the dot and dot-dot entries (this is true of both UNIX and MS-DOS) and therefore can never be totally empty. For files this means that zero-length files still own at least one cluster. Special devices are the only inodes for which we might safely and quickly optimize away this cluster, by dynamically generating impossible cluster numbers and modifying the file system utilities to recognize these directory entries.

This simple scheme no longer worked when we added the RockRidge extensions to the ISO file system. The problem is with symbolic links. In ISO a symbolic link is entirely defined in the directory entry, it does not have any associated disk space. We could have used extension-specific knowledge, such as looking into the extra

information provided by the RR records that include an inode number. Instead, we took an approach that should still work with some other future extensions to the ISO standard. The high bits of the inode number are the extent, and for regular files and directories, the low bits are zero. For symlinks, the high bits are the extent of the directory where the symlink lives, the low bits are a function of the directory entry (e.g. the offset in the extent). Any given ISO extension need only tell if this is a symlink or not, the existing code will build the inode number appropriately.

3.4. Hard/Symbolic Links

MS-DOS has neither symbolic links nor hard links. Implementing symbolic links was simple. We just took one unused bit in the attribute byte of the directory entries to mark the entry as having our special extensions, and another bit in the extension bytes to indicate a symbolic link. We store in the link's cluster the file name the link is pointing at, just like FFS. For MS-DOS the file is a regular file of length equal to the link name, and the content of the file is the link name itself.

Short FFS symlinks can be optimized by using the direct block fields of the inode to store the file name this is a link to, avoiding one disk access while doing expansion of symbolic links. DosFs has no inodes and cannot use this optimization. Notice, however, that in terms of disk accesses DosFs is always at par with the optimized case. FFS must access the directory entry, the inode, and (possibly) the link name. DosFs must access the directory entry and (always) the link name.

Hard links are more difficult. One approach is to just build another entry that points to the same starting cluster. This creates two problems: uniqueness of inode numbers is lost, and there is no obvious way to keep (on permanent storage) a reference count of the number of entries that point to the same inode. The first problem means, for instance, that size and other attributes that are stored in a directory entry are not kept coherent, unless the entry that is a hard link actually points back to the original entry it is a link to. Since a cluster can hold a number of directory entries this variation can still perform well, for the higher probability of finding the pointed-to entry in memory. But more importantly, the second problem means that there is no way to know when an inode can actually be released. One solution to this problem is to use more of those 10 reserved bytes to hold a hard link count.

Note that these issues do not arise on ISO CD-ROM file systems for the simple reason that the file system is immutable. The RockRidge extensions do support symbolic links, and they also provide reference counting. Implementing hard links and making a mutable ISO file system is therefore feasible.

For expediency, we chose to implement symlinks but not hard links. We needed symbolic links to handle the file system structure with more flexibility, especially when DosFs is the root. We did not find any mandatory reason to have hard links, so we left the issue open for future extensions.

3.5. NFS Access

Exporting a (local) file system via NFS is not difficult. The NFS layer gives to client machines a *handle* on Vnodes, the local file system provides functions to build such a handle and map it back to a Vnode. FFS handles contain just the inode number and a generation number for coherency purposes. Neither ISO nor MS-DOS have inode tables separate from directory information, we could not use just the inode number as is done in FFS. If the inode is not in the cache we need to refer back to some directory entry to recover size, modes and timestamps of a DosFs or ISO inode. This is also true with the RockRidge extensions to ISO.

The handle that is exported to remote nodes in NFS has a predefined size, but fortunately it is opaque. No application appears to make special use of the content of an FFS handle. We added two more fields to the FFS file handle scheme, the cluster number of the directory that contains an entry for the given inode, and the offset of this

entry. If the inode is not in core it can be reconstructed by recovering the original directory, checking its coherency, and looking up the directory entry. Note that protection issues are no different when access is via NFS than when access is local.

3.6. Other extensions

When we first tried to use DosFs as a root file system on a DEC Alpha workstation we discovered a small number of other problems. To begin with, we had to modify both the boot programs and the Mach default pager [3] to understand the MS-DOS file system structures. The first must be able to find, read in, and transfer control to the boot image. The second has similar needs with respect to the UNIX server image. In addition, it might have to use a DosFs file system for paging purposes.

We then found a number of instances in filesystem-independent UNIX code where FFS data structures and semantics were assumed. For instance, mounting of the root file system itself was special cased for FFS and NFS and so was the checking of special device close operations against mounted file systems. Even the shutdown of the system assumed the root was an FFS file system. We also saw no reason why the root file system should not be un-mounted, e.g. on clean shutdowns of the system.

Two more extensions were necessary to get the system up in multiuser mode. The first and most obvious, in hindsight, was the need for special devices. We used the *mode* extension field to mark a file as a UNIX special device and stored the major/minor information in the reserved bytes. Special devices are the directory entries that make the most use of the reserved bytes in MS-DOS directory entries; all 10 bytes are used up.

The second, slightly less obvious, was the need for *setuid* programs. CMU UNIX runs the single user shell not as *root* but as the less privileged user *opr*, therefore we just could not do much without encountering a protection block. Once again, we used the *mode* field for this information, making it almost identical to the FFS *i_mode* field. The only difference is that DosFs does not support the special mode value used by *badsect(1)* for creating bad block files. There is no need for this, the FAT defines special entry values to mark unusable clusters.

4. Tools

While developing DosFs we found the need for tools to accomplish a variety of tasks, such as file system construction, check and repair, bad block handling, analysis, and tuning. We patterned a small set of tools after the well known UNIX ones: *dosmkfs*, *dosfsck*, *dosbadsect*, *dosdumpfs*. All of the tools are considerably simpler and smaller than their FFS counterparts. The *dostunefs* program is actually just a compaction tool that reallocates clusters on disk to keep file/directory data contiguous for best performance. We hardly ever use it. A similar effect can be obtained simply with a *dump/dosmkfs/restore*, but with *tunefs* there is no need for a separate tape or temporary disk partition.

Once we started to use DosFs as root partition we decided to actually integrate *dosfsck* back into *fsck*, which can now handle either type of file system, including support for the so-called "preen mode". This is the way *fsck* operates non-interactively, e.g. during automatic reboots. There are only two very embarrassing cases where *fsck* will not automatically repair the disk: when an allocation chain in the FAT has a loop and when two allocation chains merge. FFS and *fsck* have many more mutual understandings of accidents that "should not happen". Another strange property of the DosFs *fsck* is that it is CPU-intensive, not I/O intensive. Most of the time is spent verifying the FAT allocation chains in memory, I/O operations to read directory entries take little time.

5. Optimizations

A number of optimizations came to us for free, by using the UNIX buffer cache for I/O. These include the caching of I/O buffers, read-ahead for sequential file accesses, and delayed-writes. Other optimizations were trivial to add, such as caching of name lookups. Handling of the FAT table was simplified by its relatively modest size. Based on our SCSI traces, and on McVoy's work on extent-based performance [9], we implemented large I/O operations.

The effects of I/O buffer caching are illustrated in Section 7, by comparing native MS-DOS and DosFs small file accesses. On a well-tuned UNIX system, cached read and write operations should perform almost at memory copy speed. For illustration purposes, we have intentionally avoided the use of any of the many native MS-DOS extensions that would perform file caching. For instance, Microsoft provides the SmartDrive optional driver which Windows 3.1 installs by default. With this proviso, read and write operations that hit in the cache are from 3 to 15 times faster in DosFs than native MS-DOS.

The benefit of delayed writes is illustrated by the `crtdel` test, where a relatively small file is created, written and deleted a number of times. The performance gain over the uncached case is up to a factor of 40. If we disable read-aheads, the elapsed time for the sequential, uncached read of a 5 MB DosFs file in 2 KB blocks increases 30%. The effective bandwidth drops from 480 KB/sec to 340 KB/sec.

The `crtdel` test reveals the effects of one interesting difference between FFS and DosFs. In FFS, directory entries and the inode table must be kept coherent in spite of potential power failures. When creating a new directory entry FFS incurs a cost of three writes. The first one is to the inode table, the second to the directory entry, and the last again to the inode table to set the reference count properly. Hard links require a proper handling of the reference count field. In DosFs we only need two writes, one to the directory entry and one to the FAT. This is still true even if we implemented hard links, because the reference count would have to be part of the directory entry itself, not of a separate inode table.

Another optimization is caching of name lookups, which also came to us for free since the name cache operates on Vnodes and is therefore filesystem-independent. The effects of this optimization can be partially quantified in the `open` tests, where a file is repeatedly opened and immediately closed. FFS and DosFs get a relatively better score on the longer pathnames than MS-DOS, but the times show that MS-DOS is also doing name caching. If we disable the namecache DosFs incurs a 15% higher cost per component. Note that in this test the read operation necessary to reconstruct the inode will always hit in the buffer cache. If the system is loaded the read might miss, and the price will be much higher.

We decided to keep the FAT entirely in memory, to speed up many file access operations. The maximum size of a FAT is 128 KB, which would describe a 512 MB partition at a cluster size of 8 KB. In practice, most MS-DOS disk partitions are much smaller than this, for instance on a floppy the FAT is approximately 4 KB and on a 60 MB disk partition of 2 KB clusters the FAT is 60 KB. In terms of main memory consumption the FFS inode cache alone is much more expensive (in our system) than the DosFs FAT table. In the Mach 3.0 system this memory is pageable. If we find that keeping the FAT in memory creates problems we can take advantage of the I/O buffer cache and reduce this memory cost to a user-specified number of I/O buffers per mounted file system. If frequently accessed, the FAT blocks will be found in the buffer cache. It is difficult to isolate the performance effects of this optimization, because the occasional I/O operations to recover the FAT would be unpredictably intertwined with the other I/O operations.

We applied other minor optimizations in the FAT handling code. For instance, every DosFs Vnode has a lookup hint that is used when mapping file offsets to disk clusters. We do not have to scan the entire allocation chain when

doing sequential reads. We also optimized the rewriting of the FATs since they tended to be a noticeable cost, especially on the slow floppies. On each sync(2) call we rewrite (if modified) only the primary FAT, the alternate FATs are rewritten only when the disk is unmounted. If we encounter a power failure the status of the disk will be as of the last sync(2) call, and dosfsck takes care of rewriting the alternate FATs.

Our disk block allocation algorithm is extremely simple and similar to the one used by MS-DOS [1]. When extending a file we scan forward in the FAT starting from the last block of the file. If we do not find a free block, then we scan backward from the last block of the file. When creating a new file or directory we start from (an estimate of) the lowest numbered free block in the FAT. The first two rules tends to enforce sequentiality and clustering, the third tends to accumulate in-use blocks at the beginning of the disk, reducing seeking. The algorithm is extremely simple, and extremely effective if executed sequentially. It is not executed concurrently in MS-DOS, or for the most part, in our UNIX workstations.

The optimization that had the single most visible payoff was to perform I/O in sizes larger than the file system's cluster size. This is possible because the block allocation algorithm is very effective in keeping a file's data contiguous. We can essentially chose any size for I/O and use it as the equivalent of the FFS logical block size. The DosFs logical block size is specified by the user at mount time. The default value of 16 KB was chosen because it is the knee in the performance curve across several disks, and across three types of workstations, but we make no claims as to its generality and applicability. The implementation is simple, we use a special version of *bmap()* that returns the start and size of the contiguous chunk of disk that contains a given cluster.

One last optimization we adopted from MS-DOS: we do not force the users to discover optimizations on their own, unless they are really obvious. This is in sharp contrast with the FFS philosophy. In [8] it was advocated that users would know the characteristics of all disks and even the speed of their processors relatively to the disks they used. Users would otherwise perform experiments to define the correct values for the file system parameters, and use *tunefs(1)* to optimize them. There is only one tuning parameter in DosFs, the logical block size for large I/Os. This is optionally decided at mount time, has an appropriate default, and the meaning is more intuitive than most FFS parameters (and/or combinations thereof).

6. Disk Utilization

In this section we analyze the overheads and the costs for metadata in DosFs, and we compare them with FFS. We have performed a simple experiment involving the storing of a set of files on a floppy disk, and we have derived the formulas that characterize the space requirements of the two file systems.

<u>Allocation Size</u>	<u>Waste</u>	<u>FFS free</u>	<u>DosFs free</u>	<u>Difference</u>
512	50 KB	326 KB	422 KB	+29%
1024	105 KB	273 KB	368 KB	+34%
2048	220 KB	138 KB	260 KB	+88%
4096	475 KB	-86 KB	4 KB	n/a

Table 6-1: A Simple Utilization Test: Moving Data To A Floppy Disk

The data set included two software packages in sources, objects, documentation files and man pages for a total of 978,634 bytes in 194 files and 7 directories. The floppy media has a formatted capacity of 2880 sectors, or 1,474,560 bytes. Table 6-1 illustrates the results of unpacking the data from a tar file onto the floppy file system. Four cases each for FFS and DosFs are presented, with fragment/cluster sizes ranging from 512 to 4096 bytes. The FFS block size was kept constant, 4096 bytes. We report the amount of free space after the unpacking (a negative

value indicates the amount of data that did not fit), as reported by `df(1)`. The FFS entry at 4096 bytes is misleading since not all of the data fit on disk. DosFs leaves from 30% to 80% more free blocks than FFS, depending on the allocation size.

The first column reports the number of bytes wasted due to fragmentation. Suppose we put a one-byte file in a single directory on the floppy. We will need one disk block for the directory, and one for the file. Most of the bytes in the two blocks are wasted, in DosFs we really needed 33 bytes and we used instead 1024 bytes, or more. We computed the fragmentation by looking at each file and directory and evaluating the extra bytes on disk that were allocated and unused because data did not fill a block. The space wasted is approximately the same in the two filesystems and originates for the most part from files. This measure is important in two respects. It shows that an improper allocation block size can easily cost 40% of the available space on disk. It demonstrates that Extent Based allocation, which is equivalent to even larger allocation sizes, is an expensive proposition for typical UNIX files.

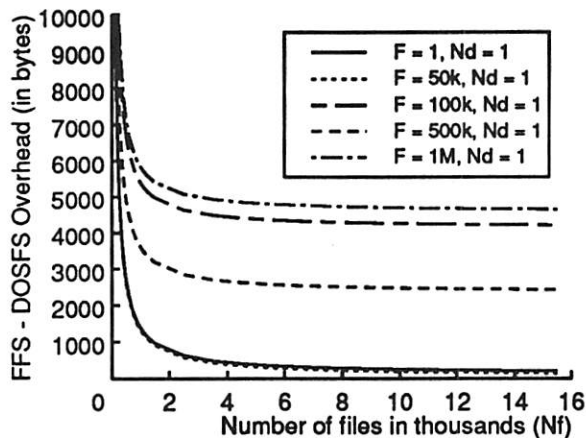


Figure 6-1: Metadata Costs: FFS Over DosFs

Let us now look more closely at the overheads incurred for meta-data in the two file systems, by comparing the total per-file disk costs for a file of length F in a file system of fragment/cluster size A . In the formulas, $\text{ceil}(X, Y)$ is the function that rounds up X to the nearest Y multiple. DosFs will use

$$\text{DosFsCost} = D + (B * \text{Ceil}(F/A, 1)) + \text{Ceil}(F, A)$$

$$D = \text{Ceil}(Nd * 32, A) / Nd$$

bytes on disk, where Nd is the number of files in the same directory and B is the number of bytes per cluster in the FAT, e.g. 2 for a disk and 1.5 for a floppy. The three terms of the sum represent the cost D in the directory entry, the FAT, and the file itself. The FAT cost is paid once at file system creation time, and with the exception of the root directory all other costs are dynamic allocations. The fixed, upfront cost for DosFs in the (best) floppy case is 2 KB, in a 60 MB disk partition this is 64 KB.

For FFS, we arrive at the following expressions

$$\text{FFSCost} = D + 128 + \text{Ceil}(F/(8 * A), 1) + C / Nf + \text{Ceil}(F, A)$$

$$D = \text{Ceil}(\text{Sum}(\text{Max}(12, 8 + \text{namelen}(i))), A) / Nd$$

Where Nf is the total number of files on disk, and C is the fixed cost in cylinder groups and replicated superblocks. For a floppy this cost is at minimum 104 KB in three cylinder groups, for a 60 MB disk partition it is 1.4 MB. The five terms of the sum are the amortized directory entry cost D , the inode, the allocation bitmap, the

amortized fixed cost, and the file itself. For FFS all but the directory entry and file data costs are paid at disk creation time. The formula above is only valid for (relatively) small files, for very large files we must also add the costs in double and triple indirect blocks.

Figure 6-1 shows what happens as we add more files to the file system. The curves assume a given file length F , and a fixed number of files per directory. Note that up to a file length of 64 KB FFS does not use indirect blocks to indicate the blocks that belong to files, it uses the direct block list in the inode which is a pre-paid cost. This fact is illustrated in more detail in Figure 6-2, where we vary the size of the average file in a file system with 6,000 files and an average of 20 files per directory. In both Figures the fragment size is 512 bytes and the logical block size is 4 KB, in a 60 MB partition. The ramps are the points at which FFS must allocate a new block for the (indirect) block list. Once that cost is paid DosFs has a lesser advantage, but the DosFs costs still grow at half the speed of the FFS costs. This is because FAT entries are 16 bits while FFS disk block numbers are 32 bits.

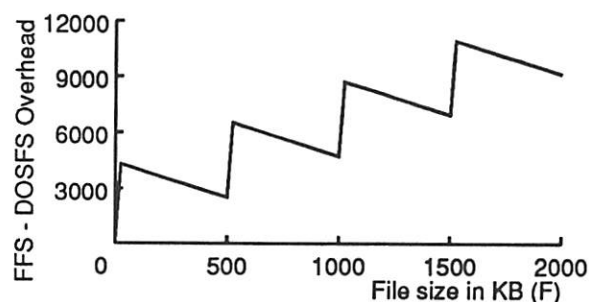


Figure 6-2: File System Overhead In Bytes, As A Function Of The File Size

If we look at directory entries alone, FFS would score better than it does overall. DosFs uses a fixed 32 bytes per entry, including information that FFS stores in the inode table. FFS instead uses a minimum of 8 bytes plus the file name, and will typically be able to fit more directory entries per block. In Figure 6-3 we plot the directory entry costs D for the two file systems, and their difference. By growing the number of files per directory the initial cost is amortized and the remaining cost is dominated by the file name length (we have assumed file names of length 12).

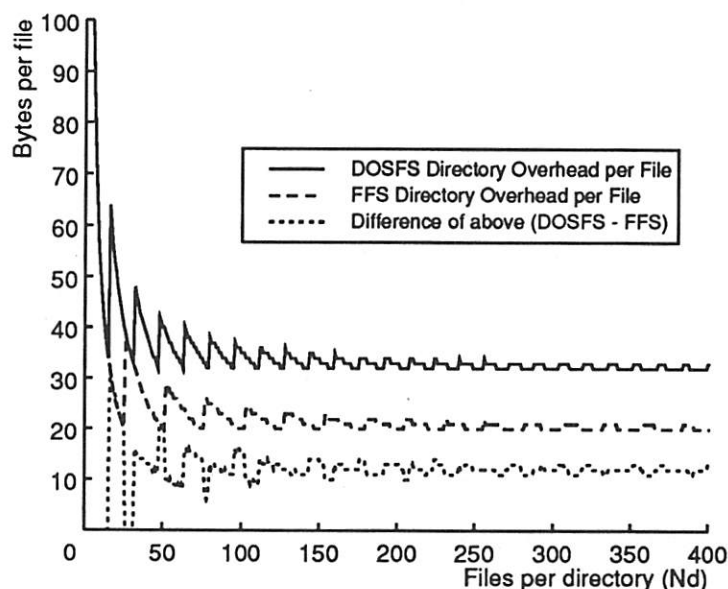


Figure 6-3: Costs Of Directory Entries

Note that the block allocation algorithms used in the two file systems have no effects on disk utilization, but they do affect performance as indicated in Section 7.

7. Performance Evaluation

John Ousterhout published, as part of the Sprite OS sources, a set of file system tests [11], which we use to summarize the performance of our file system at the micro-benchmark level. The `open` test opens (and immediately closes) the same pathname a large number of times, and reports the average time per iteration. The `crtldel` test repeatedly creates, writes to, and deletes a file. The `read` test reads from `stdin` in 16 KB sizes, until the end of the file. The file is then reset and the cycle is repeated at least 100 times, for timing purposes. Similarly `write` writes the given number of KB to `stdout`, in no more than 16 KB sizes. This is also repeated at least 100 times.

Most of the well-known macro-benchmarks are UNIX-specific and do not have an identical implementation and meaning under MS-DOS and UNIX. This is true for the Andrew benchmark [4], for which we do not have a native MS-DOS version. Similarly for the unpacking of the tar file described in Section 6; we can only compare the times to do the unpacking between FFS and DosFs. The Iostone test is a synthetic benchmark that models a timesharing load. The test was written by Arvin Park and Jeff Becker at UC Davis, and it is based on the traces John Ousterhout collected on a general-purpose machine at UC Berkeley [10]. For MS-DOS, we report on typical operations a user would perform on a large number of files, such as copying or deleting file trees.

The test machine is a Gateway 2000 486/33C running Mach 3.0 (MK83+UX42), with a 33 Mhz Intel i486 processor, 16 MB of main memory and a 400 MB Maxtor SCSI disk interfaced via an Adaptec 1540c SCSI HBA. Effective memory bandwidth is about 10 MB/sec. The disk is split into a 60 MB initial MS-DOS partition and the remaining disk space for UNIX partitions. Both the MS-DOS and FFS partitions are over two years old. The MS-DOS partition is about 40% full, the cluster size is 2 KB. The FFS partition is about 60% full, fragsize is 512 bytes, block size is 4 KB, rotldelay is 4 ms, maxcontig is 1, there are 62 cylinder groups. Using fresh file systems would lead to a more repeatable experiment, but using old file systems shows what happens to the performance numbers when the file systems age. The discrepancy between our results and the results presented in [7] seems to indicate that DosFs has a better degradation than FFS.

<i>Test</i>	<i>Native MS-DOS</i>	<i>FFS</i>	<i>DosFs</i>
open "foo"	0.50 ms	0.75 ms	0.75 ms
open "a/b/foo"	0.82 ms	0.82 ms	0.82 ms
open longpath	1.16 ms	0.95 ms	0.95 ms
crtldel 0 KB	32 ms	3 ms	3 ms
crtldel 10 KB	521 ms	36 ms	20 ms
crtldel 100 KB	3671 ms	107 ms	87 ms
read 1 KB	0.1 MB/s	1.1 MB/s	1.1 MB/s
read 8 KB	0.5 MB/s	2.2 MB/s	2.5 MB/s
read 100 KB	1.2 MB/s	3.0 MB/s	3.3 MB/s
write 1 KB	0.1 MB/s	1.3 MB/s	1.3 MB/s
write 8 KB	0.5 MB/s	3.7 MB/s	3.7 MB/s
write 100 KB	0.5 MB/s	5.8 MB/s	7.4 MB/s

Table 7-1: John Ousterhout's Basic File System Tests.

The results of the micro-benchmarks are illustrated in table 7-1. DosFs is faster than FFS on cached read and write operations, but this is simply the result of the way data is moved between the UNIX server and the user

process. Mach's Virtual Memory works best on large amounts of data, and in the case of DosFs it can move the entire 16 KB segment by remapping it at once. In the FFS case, data is split across 4 buffers and disjoint virtual addresses.

The lack of caching and/or prefetching in MS-DOS makes it run slower, on average, than the UNIX file systems. We modified the native MS-DOS *read* test to read different amounts of data with various blocking factors; the best bandwidth was 1.2 MB/sec when reading a 100 KB file in 16 KB blocks. In order to explain the results of the *open* test we must assume that native MS-DOS is doing some caching anyway. This was confirmed with the Mach 3.0 MS-DOS emulator.

To evaluate the performance on uncached operations we use the *read* and *write* tests above, on much larger files. The tests include both networked and local cases. Read and write performance of DosFs over NFS are nearly identical to the FFS performance, as indicated in table 7-2. The client was a Decstation 5000/200, the server was a DEC Alpha workstation, both running Mach 3.0. The user-to-user bandwidth (measured by *ttcp*) between the two machines over a UDP transport was 860 KB/sec. Both the NFS and the networking code in the UNIX server are old, and these numbers show it.

<i>Test</i>	<i>FFS</i>	<i>DosFs</i>	<i>MS-DOS Native</i>
Uncached read 5MB	170	850	1200
Uncached write 5MB	178	382	624
NFS read 5MB	377	379	n/a
NFS write 5MB	749	749	n/a

Table 7-2: Uncached Data Transfers, For Local Disk And Over NFS, In KB/sec

The UNIX server has disabled mapped files, therefore we incur data movement overheads that make local performance lower than in a monolithic kernel. To quantify the maximum disk performance, we wrote a program that accesses the disk directly through the Mach 3.0 kernel. This is the closest equivalent of the raw character device for traditional UNIX. This program uses multiple threads to guarantee that a request for the next sequential block is always ready in the disk driver queue by the time a request completes. This was verified with a SCSI bus analyzer. On the test machine this program obtains a 1.5 MB/sec read rate at a block size of 16 KB. Higher block sizes obtain bandwidths in excess of 1.6 MB/sec¹. By contrast, a *dd(1)* of the raw device via the UNIX server only obtains 1.2 MB/sec, which indicates a 20% overhead in the UNIX emulation code along the I/O path.

In table 7-2 we indicate the results of reading or writing a 5 MB file for each of the three file systems. The buffer cache size was set at 1.3 MB and the file systems were unmounted between each of five test runs. We report the best results, but there is little variation even if the machine was running in multi-user mode on a regular Ethernet. Extra data copies make DosFs slower than MS-DOS. But the most remarkable result here is the five fold speedup of DosFs over FFS. This result can be explained in the same way as in [7]: DosFs performs I/Os on a file that is contiguous, whereas FFS allows for presumed rotational delays and interleaves blocks on the disk. Every track is only 1/4 utilized by FFS, losing most of the benefits of the track-level caching done by the disk. The speedup is a little higher than four because the FFS file system is not empty and occasionally FFS cannot allocate its blocks optimally. DosFs is still capable of reaching optimal allocation in its partition, despite the utilization.

We obtain another important result if we disable the large I/O optimizations, e.g. if we force DosFs to perform I/O at the 2 KB cluster size. The DosFs bandwidth becomes 480 KB/sec, almost three times higher than FFS which

¹Poor cabling prevented us from making use of the faster SCSI synchronous transfer mode.

does I/O in 4 KB blocks. The disk compensates for the small access size by read-caching an entire track, but it cannot compensate for the extra seeks that FFS incurs. Adding track caches on the disk drives is quite common with SCSI disks.

Table 7-3 shows the results of our macro-benchmarks. The modified Andrew benchmark is commonly used to simulate the load of a programming environment. The test has five phases which mainly involve the creation of various directories (phase I), copying of files (phase II), doing many `stat(2)` over the file tree (phase III), sequentially scanning all files (phase IV) and performing some compilations (phase V). DosFs is faster than FFS in all phases, the larger gains are obtained in the phases that actually access file data, e.g. phases IV and V. Among all the tests we ran, this is the one with the least controlled execution environment. It uses scripts and system programs, for instance, which on the test machine are paged in from a different partition but on the same disk. The distance between the DosFs partition and the FFS partition from system data and code is not the same, and DosFs is penalized. To use this test effectively we would need a different test setup.

We report three cases for the Iostone test, in each one we used a different number of instances of the test, running in parallel directories. Each instance uses approximately 5 MB of disk space, to overflow the buffer cache. We selected this test to get a worse-case behavior from DosFs, because it invalidates the assumptions about a single-user load. When we checked the block allocations for the larger files we found them very scattered, as expected. But DosFs is still from 13% to 20% faster than 4.3 FFS when large I/O operations are enabled. If we disable large I/Os (times in parentheses) DosFs becomes at most 20% slower than FFS.

<i>Test</i>	<i>Native MS-DOS</i>	<i>FFS</i>	<i>DosFs</i>
Andrew - Phase I	n/a	0.5	0.5
Andrew - Phase II	n/a	9	8
Andrew - Phase III	n/a	8	6
Andrew - Phase IV	n/a	12	9
Andrew - Phase V	n/a	49	40
Iostone-1	n/a	233	195 (287)
Iostone-2	n/a	654	550 (700)
Iostone-3	n/a	1089	963 (983)
Unpack Tar File	n/a	187	150
Copy Windows	48	76	58
Delete Windows	11	5.4	1.2

Table 7-3: Results For Selected Macro-Benchmarks, All Times In Seconds

We had expected DosFs to be as much as twice as slow as FFS in the Iostone test, due to the smaller I/O size (2 KB versus 4 KB). This is not the case, and the explanation is not immediate. The majority of the files used by the test are small, only 12 of the 404 work files are larger than 16 KB, 192 files are one sector or less, and 40 files are between 4 KB and 16 KB in size. Even the larger files are not totally fragmented. Two distinct phenomena help explain the results of this test. DosFs keeps the many little files close to each other, making better use of the disk-level cache, and when using the large buffers DosFs has a higher hit rate in main memory.

We measure the unpacking of a 1 MB tar file to a floppy disk for two reasons. It is an example of a common activity in UNIX, and the floppy is extremely slow when compared to a disk and does not have any read or write caches to help performance. Here the FFS optimizations show much gain, the difference versus DosFs is no longer a factor of two as in the uncached write test but a much smaller 25%. The test involves creating many files, therefore FFS pays the extra costs discussed in the `crtidel` test above.

For the last two tests we also have native MS-DOS times. The first one is a recursive copy of a relatively large tree of MS-DOS system files, the Windows 3.1 directory. The tree is larger than the buffer cache, and the test was performed on a cold cache to guarantee pure uncached behavior. The results confirm the trend of the previous uncached micro-benchmarks, but the gap between systems is reduced. Deleting the Windows tree shows once again the advantage of delayed writes, as was partially indicated in the `crtdel` test.

There is one optimization we did not apply to DosFs which is instead present in FFS: caching of inodes. The UNIX server, for instance, uses an in-memory inode table of about 1,200 FFS inodes. Even if released (usually because of namecache spills) FFS inodes stay around and can be reused without disk accesses. None of the tests above would be strongly affected by this optimization, but for instance the traversal of the whole file system name space probably would. Caching could be done at the Vnode layer, provided it benefits all file systems.

8. Related Work

Sun provides the *pcfs* file system in its UNIX offerings, which is a read/write Vnode based file system for MS-DOS disks, like our DosFs. Next has a similar file system in the NeXTStep product. Recently there was a posting on alt.sources of a "Filesystem Survival Kit" which provides the ISO and MS-DOS file systems for System-V UNIX systems. These file systems are strictly MS-DOS specific, and offer no protection extensions, or symlinks. They do not provide performance optimizations, and cannot be used as the root file system.

The SunOS 4.1 code lets the user change floppy disks without using `umount/mount`. It does this by serializing all operations and performing consistency checks between in-memory and on-disk data. As a result, the code is less effective at caching. NeXTStep uses the hardware to detect a change in floppy media, and auto-mounts floppies on insertion. The use of hardware detection should enable more optimizations.

In the non-UNIX world, IBM's OS/2 offers multi-user extensions to the PC-DOS FAT-based file system which they call *extended attributes*. These can support more than just protection information; they can be used for presentation purposes for instance. Extensions by the same name are defined in the ISO 9660 standard, and serve similar purposes.

Our optimizations for uncached accesses are similar, but unrelated to the optimizations described in [9, 7] for Extent Based File Systems. DosFs adds to McVoy's work the ability to specify the extent size at mount time. The implementation of large operations is similar to the BSD 4.4 *cluster* idea, the *bmap()* changes are almost identical. But while our optimizations are specific to DosFs, the BSD 4.4 clusters are not filesystem-specific and can be used by any file system code. We call our *bmap()* only from within DosFs code, in 4.4 we would call the DosFs *bmap()* from the filesystem-independent clustering code.

9. Conclusions

With DosFs we have demonstrated that it is possible to provide full UNIX semantics in a file system with simpler data structures than traditionally used in UNIX file systems. Removing inodes provides better disk utilization, better disk/power failure behavior, and eliminates many synchronous write operations in the handling of meta-data. The only cost is a more cumbersome and slow implementation of hard file links.

In the implementation of DosFs we have used a new block allocation algorithm, derived from MS-DOS [1]. The algorithm obtains both contiguous allocation of disk blocks and compact disk utilization. DosFs is both faster than 4.3 BSD FFS and more efficient in disk space. The performance gains are equivalent to Extent Based allocation, both without the high fragmentation costs. DosFs allocates small blocks, but close to each other. In this way it can take advantage of disk-level read and write caches, and obtains a three fold performance improvement over FFS.

Moreover, DosFs takes advantage of the contiguous block allocation to perform large size I/O operations and obtains a five fold speedup over FFS.

An important factor in the DosFs performance profile is the effect of seek times, or more precisely the time spent by the disk to *service* a given I/O request. Today's disks use a variety of multi-track caching strategies that make the old FFS model [8] unrealistic. Unfortunately, the service time has become very difficult to model. In our SCSI analyzer traces we observed large variations in service times, anywhere from 3 milliseconds to 45 milliseconds, on the same disk, and not at all a simple function of the distance between disk blocks. When the disk drive uses a write cache, not only the distance, but the type of access and previous history affects the service time. Finally, the zoning techniques used in recent drives add another variable to the already complicated model. All of these factors contribute to make DosFs' performance better than FFS, even in cases where we did not expect it.

The only deficiencies in the MS-DOS permanent data structures are in handling large files and large disks. Large files create two problems, both related to the choice of a linked list to represent the clusters of a file. In the first place, is it impossible to represent files with holes, all clusters must be allocated by DosFs and the "holes" truly must be zeroed for security reasons. The programmer does not have to issue the intervening writes, but there is no way to indicate in a FAT entry that the block ought to be zeroed when accessed. FFS uses a tree as its data structure, and can easily mark the holes as missing blocks, to be zeroed on access. But files with holes are not commonplace, and a copy of the file will fill the holes anyway. The second problem is we must traverse the whole list to get to the last cluster of a file. We have argued that it is feasible to cache the entire FAT in memory so that the cost is just in memory accesses, and much less than the case where FFS must actually fetch an indirect block from disk.

Large disks create the problem of representing cluster numbers, which are currently 16 bit integers in the permanent data structures. It is possible to use a new FAT type, with 32 bit entries. It is not possible to transparently change the format of the directory entries to hold a 32 bit starting cluster instead of the 16 bit one. A temporary solution can be to grow the cluster size, but this can only gain a few more bits, if the cluster size becomes 32 KB the fragmentation (most MS-DOS files are small) is unacceptable. If we ignore MS-DOS compatibility this problem disappears.

9.1. Status and Acknowledgements

The code described in this paper has been integrated in the CMU sources (releases UX42/43), and is available for general use at CMU and elsewhere. Our shepherd Margo Seltzer went way beyond the call of duty in helping us put the paper in shape. Mike Dryfoos, Mahadev Satyanarayanan, Robert Baron, Daniel Stodolsky, Howard Gobioff and Peter Dinda gave us many helpful comments on earlier versions of this paper. Thanks for her patience, and good luck to Kelly.

References

- [1] Ray Duncan.
Advanced MSDOS Programming.
Microsoft Press, 1986.
- [2] David Golub, Randall Dean, Alessandro Forin, Richard Rashid.
Unix as an Application Program.
In *Proceedings of the USENIX Summer Conference*, pages 87-95. USENIX, Anaheim, CA, June, 1990.
- [3] David Golub and Richard Draves.
Moving the Default Memory Manager Out of the Mach Kernel.
In *Proceedings of the USENIX Mach Symposium*, pages 177-188. USENIX, Monterey, CA, November, 1991.

- [4] Howard, J. et al.
Scale and Performance in a Distributed File System.
ACM Transactions on Computer Systems 6(1), February, 1988.
- [5] ISO 9660 : 1988 (E).
Information processing - Volume and file structure of CD-ROM for information interchange.
International Organization for Standardization, 1988.
- [6] Kleiman, S. R.
Vnodes: An Architecture for Multiple File System Types in Sun UNIX.
In Proceedings of the Summer 1986 USENIX Conference, pages 238-247. June, 1986.
- [7] Margo Seltzer, Keith Bostic, Kirk McKusick, Carl Staelin.
An Implementation of a Log-Structured File System for Unix.
In Proceedings of the USENIX Winter Conference, pages 201-220. USENIX, San Diego, CA, January, 1993.
- [8] Kirk McKusick, William Joy, Sam Leffler, R. Fabry.
A Fast File System for UNIX.
ACM Transactions on Computer Systems :181-197, August, 1984.
- [9] Larry McVoy and S. Kleiman.
Extent-like Performance from a Unix File System.
In Proceedings of the USENIX Winter Conference, pages 33-44. USENIX, Dallas, TX, January, 1991.
- [10] John Ousterhout, et al.
A Trace-Driven Analysis of the UNIX 4.2 BSD File System.
In Proceedings of the Tenth Symposium on Operating Systems Principles, pages 96-108. December, 1985.
- [11] John Ousterhout.
Why Aren't Operating Systems Getting Faster As Fast As Hardware?
In Proceedings of the Summer 1990 USENIX Conference. June, 1990.
- [12] Richard Rashid, Gerald Malan, David Golub, Robert Baron.
DOS as a Mach 3.0 Application.
In Proceedings of the USENIX Mach Symposium, pages 27-40. USENIX, Monterey, CA, November, 1991.
- [13] Rock Ridge Interchange Protocol, Version 1.
An ISO 9660:1988 compliant approach to providing adequate CD-ROM support for Posix file system semantics.
Rock Ridge Technical Group, 1991.
- [14] Satyanarayanan, M.
Integrating Security in a Large Distributed System.
ACM Transactions on Computer Systems 7(3), August, 1989.

Dr. Forin is a Research Computer Scientist at Carnegie Mellon University, working on operating systems. He received the B.S. degree in Electrical Engineering in 1982 and the Ph.D. degree in Computer Science in 1987, both from the University of Padova, Padova, Italy.

Gerald R. Malan is a Research Programmer at Carnegie Mellon University, and has been working on the Mach project for the last four years. His research interests include multiple operating system personalities, remote debugging, file systems, and object oriented system composition. Mr. Malan received his B.S. in Computer Engineering from Carnegie Mellon University in 1990.

An Overview of the NetWare Operating System

*Drew Major
Greg Minshall
Kyle Powell*

Novell, Inc.

Abstract

The NetWare operating system is designed specifically to provide service to clients over a computer network. This design has resulted in a system that differs in several respects from more general-purpose operating systems. In addition to highlighting the design decisions that have led to these differences, this paper provides an overview of the NetWare operating system, with a detailed description of its kernel and its software-based approach to fault tolerance.

1. Introduction

The NetWare operating system (NetWare OS) was originally designed in 1982-83 and has had a number of major changes over the intervening ten years, including converting the system from a Motorola 68000-based system to one based on the Intel 80x86 architecture. The most recent re-write of the NetWare OS, which occurred four years ago, resulted in an "open" system, in the sense of one in which independently developed programs could run. Major enhancements have occurred over the past two years, including the addition of an X.500-like directory system for the identification, location, and authentication of users and services. The philosophy has been to start as with as simple a design as possible and try to make it simpler as we gain experience and understand the problems better.

The NetWare OS provides a reasonably complete runtime environment for programs ranging from multiprotocol routers to file servers to database servers to utility programs, and so forth.

Because of the design tradeoffs made in the NetWare OS and the constraints those tradeoffs impose on the structure of programs developed to run on top of it, the NetWare OS is not suited to all applications. A NetWare program has available to it an interface as rich as that available to a program running on the Unix operating system [RITC78], but runs in an environment that is as intolerant of programming mistakes as is the Unix kernel.

The NetWare OS is designed specifically to provide good performance for a highly variable workload of service requests that are frequent, overlapping, mostly of short-duration, and received from multiple client systems on a computer network. In this capacity, the NetWare OS is an example of a network operating system. General-purpose operating systems, on the other hand, are designed to provide appropriate levels of service across a wide spectrum of differing workloads.

Just as a general-purpose operating system provides primitives required to act as a network operating system, the NetWare OS provides primitives required to develop most programs capable of running on a general-purpose operating system.

The differences between a general-purpose operating system and an operating system like the NetWare OS lie mostly in the handling of tradeoffs between ease of programming and fault tolerance, on the one hand, and execution speed, on the other hand.

General-purpose operating systems normally have a preemptive scheduler. This frees the programmer from involvement in the system's scheduler. This also increases fault tolerance by preventing a misbehaving program from monopolizing the system. The NetWare OS, on the other hand, is non-preemptive by design. This makes the system more responsive under load and simpler (and, therefore, faster) by reducing the number of concurrency issues, such as the locking of data structures, that programs need to manage.

General-purpose operating systems normally use hardware facilities to keep one program from incorrectly accessing storage locations of another program. This increases the fault tolerance in a system by isolating failures to separate regions of the system. The NetWare OS allows all programs to run without hardware protection. This reduces the time required to communicate between programs and between a program and the NetWare OS.

General-purpose, multi-user operating systems normally enforce a strong boundary between the kernel and user-level programs. Additionally, any given instance of a user-level program is often identified with exactly one end user for the purpose of access control. Both of these attributes contribute to the fault tolerance and security of such a general-purpose operating system. In contrast, the NetWare OS makes no formal distinction between kernel and user-level programs. This allows the functionality of the operating system to be easily extended by other programs. Moreover, the NetWare OS does not prevent any program from assuming the identity of any end user for access control purposes. In an environment in which one program may provide service to 1,000 network clients simultaneously, it is more efficient to allow the program to assume different users' identities as it services different requests.

The purpose of this paper is to describe the NetWare OS in greater detail. Section 2 introduces some terms used in the paper and provides an overview of the NetWare system. Section 3 provides more detail on the kernel of the NetWare OS, including a detailed description of the scheduler, one of the keys to the performance of the system. Section 4 describes a software approach to fault tolerance known as "server replication". We conclude with a brief discussion of some future work. For brevity, this paper does not describe the NetWare file system, network protocols and implementation, or other higher level services of the NetWare OS.

2. Overview and Terms

This section defines certain terms and gives an overview of NetWare and the NetWare OS.

2.1. General Terms

An operating system manages the basic resource sharing within a computer system. In addition, an operating system provides a suite of programming abstractions at levels somewhat higher than those provided by the hardware on which it runs [TANE92].

The kernel of an operating system manages the sharing of hardware resources of the computer system, such as the central processing unit (CPU) and main storage. It also provides for interrupt handling and dispatching, interprocess communication, as well as, possibly, the loading of programs into main storage.

The schedulable entities of a system are known as threads. A thread consists of a program counter, other hardware register contents, and other state information relevant to its execution environment.

While processing, we can distinguish between different system states as follows. Interrupt time is the state that occurs after the processor has accepted an interrupt (either hardware or software) and before the software has returned from the interrupt to the normal flow of control. Processing during interrupt time normally occurs on the stack that was active when the interrupt occurred, or on a special stack reserved for interrupt processing. Process time is the

normal flow of processing in which the kernel, operating system, and application programs run. Processing during process time occurs on the stack of the currently running thread. There is a third state that we will call software interrupt time that may occur at the end of other kernel activity [LEFF89]. During this time, the kernel may decide to perform certain activities not directly related to the original reason for entering the kernel. During this time, processing occurs on a borrowed or reserved stack.

In the NetWare OS, software interrupt time occurs only when the scheduler has been entered (see section 3.3).

In general, execution is allowed to block only during process time.

An upcall [CLAR85] is a mechanism where a (typically) lower layer of a system can signal an event to a (typically) higher layer in the system by means of a procedure call. In the NetWare OS, these upcalls are known as Event Service Routines (ESRs) or callback routines.

Assume that the main storage of a system is divided into memory objects (e.g., bits or bytes). Assume that the state of execution of a thread at a given time is summarized in an execution context that determines the value of the program counter and other pieces of state. One element of a thread's execution context is the thread's rights to access the different memory objects of the system. These access rights are enforced by the hardware of the system, specifically the virtual memory system, storage keys, or other hardware specific components. In general, these access rights include "read", "execute", "read/write", and "none". (In the NetWare OS, only "read/write" and "none" are actually used.)

Using these concepts, we are in a position to define protection domain. Two memory objects are said to be in the same protection domain if and only if for each execution context X in the system, X has the same access rights to both memory objects. Similarly, two execution contexts are said to be in the same protection domain if and only if for each memory object X in the system, both execution contexts have the same access rights to X.

In the NetWare OS there is a privileged protection domain known as the kernel domain. An execution context whose protection domain is the kernel domain has "read/write" access to all memory objects in the system.

We say that a memory object is in the protection domain of a given execution context (except the kernel protection domain) if and only if the execution context has "read/write" access to the memory object. We say that a memory object is in the kernel protection domain if and only if it is not in any other protection domain in the system. Note that protection domains induce a partition on all the memory objects and on all the execution contexts in the system.

We then speak informally of a protection domain as an equivalence class of execution contexts and the related equivalence class of memory objects.

2.2. NetWare Overview and Terms

A NetWare system (also known as a "NetWare network" or "NetWare LAN") provides for the sharing of services over a network. The two principle components of the system are clients, which request service, and servers, which arbitrate the requests and provide the service.¹ Typical services include file systems, files, printer queues, printers, mail systems, and databases. The network can either be a single subnet, or a collection of subnets interconnected by routers.

In a NetWare system, clients run "off the shelf" operating systems such as DOS, Windows, OS/2, Unix, and the Macintosh operating system. Depending on the specific client and on the service being requested, it may not be necessary to add any additional software to the client in order to participate in a NetWare system. Macintosh and Unix clients, for example, typically come with built-in networked file system client software and thus don't require additional software to obtain file service in a NetWare system.

¹In certain configurations, both the client and server may run on the same computer.

Servers in a NetWare system run the NetWare operating system, an operating system that has been designed specifically as a platform for running programs that provide data and communications services to client systems. The NetWare OS currently runs on Intel 80x86 systems; there is ongoing work to port the operating system to some of the current Reduced Instruction Set Computer (RISC) platforms.

A NetWare server consists of the NetWare OS kernel and a number of NetWare Loadable Modules (NLMs). An NLM is similar to an "a.out" file in the Unix operating system [RITC78]: it consists of code, data, relocation, and symbol information for a program. An NLM is generated by a linkage editor, which uses as input one or more object files as well as a control file. The object files are generated by assemblers and/or compilers from source files. The relocation information allows an NLM to be loaded at any address in the system. NLMs also contain unresolved external references that are resolved at load time (as described in section 3.1). Finally, NLMs may contain directives that allow them to export procedures for use by other NLMs (again, this is described in section 3.1). In this final sense, NLMs resemble shared libraries [GING89].

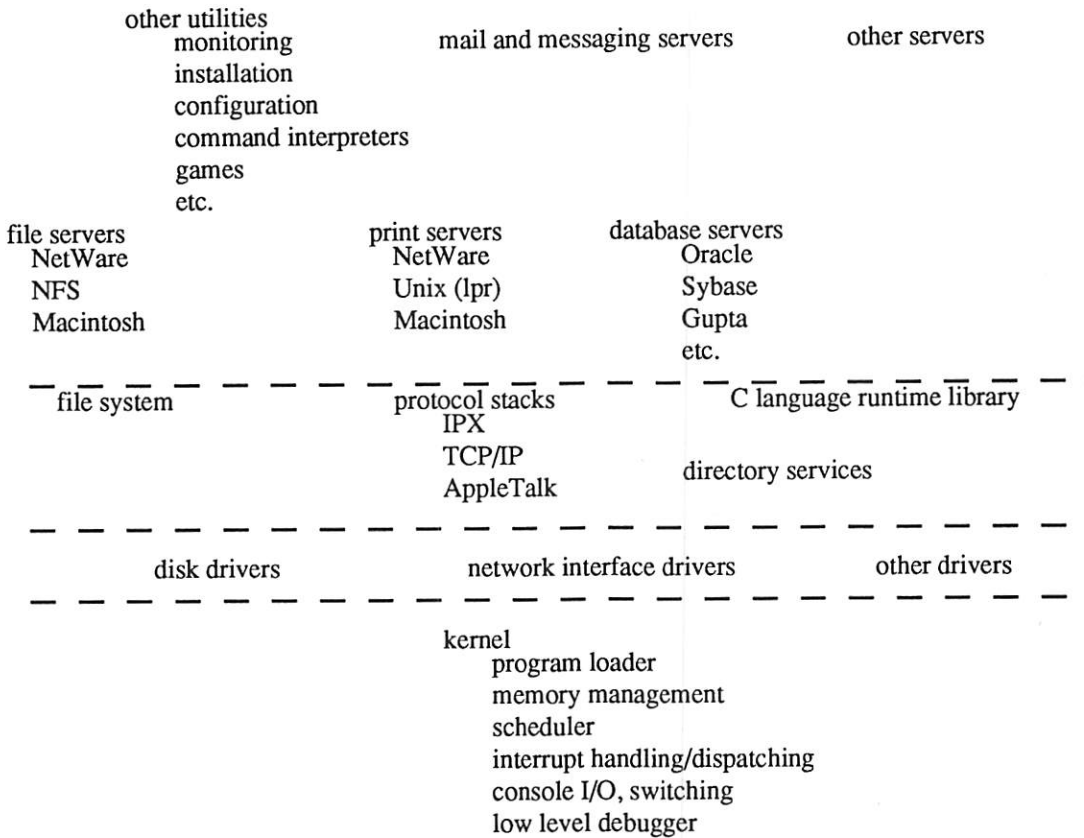
The performance requirements for NLM loading are less stringent than those for the loading of Unix a.out files because, in general, an NLM tends to run for a longer period of time than a Unix process and, therefore, does not need to be loaded and unloaded as frequently as a Unix a.out file.

The NetWare OS does not have the concept of a process. Instead, the basic units are the NLM, a protection domain, and a thread. In part, we do not use the term process because it has such different meanings in different systems. Depending on the system, there may be no memory sharing between processes, or there may be more than one process in the same memory space; there may be exactly one thread in a process, or there may be many threads in a process; there may be exactly one executable in a process, or there may be multiple executables in a process.

Although the NetWare OS has no concept of a process, we use the term "process time" (section 2.1) for what otherwise might be called "thread time".

Conceptually, there is no memory sharing between protection domains. In practice, the kernel domain has read/write access to other protection domains. Additionally, in the current version of the operating system shared memory is used in the implementation of procedure calls between protection domains (described in section 3.1).

The following figure illustrates the conceptual structure of a typical NetWare server. The dashed lines indicate the boundaries between the various layers in the system; these boundaries, like the lines that denote them, are not solid divisions because one NLM may provide service at several levels.



Outside of the kernel, each separate piece in the above figure consists of one or more NLMs. Thus, NLMs can be device drivers, higher level parts of the operating system, server processes, or utilities.

3. The NetWare Kernel

The NetWare kernel includes a scheduler, a memory manager, a primitive file system with which to access other pieces of the system during the process of booting up the system, and a loader that is used to load NLMs. In contemporary usage, the NetWare kernel might be known as a microkernel [GIEN90].

3.1. The Loader

Currently, all NLMs share the same address space, though this address space can be divided up into various protection domains.² By default, all NLMs are loaded into and run in the kernel domain. At the time an NLM is loaded, however, the user can specify a protection domain into which to load the NLM. All NLMs resident in a given protection domain have shared memory access to each other's memory. The kernel domain has read/write access to memory in all other protection domains.

The loader maintains a symbol table in order to resolve external references when loading NLMs. When the system first loads, this symbol table contains one entry for each procedure exported by the NetWare kernel.

At load time, any unresolved external references in an NLM will be bound using the symbol table. Any symbol left unbound after load time will cause the NLM load to fail. There are runtime facilities that an NLM can use to test

²In the current release of NetWare, only two protection domains are available. This is a restriction that we hope to lift.

the availability of a binding for a given symbol, to register for a notification in the event a given symbol changes its binding, and to determine the binding itself.

In addition to “importing” procedures, an NLM can “export” procedures – that is, make them available to other NLMs. Procedures exported by an NLM will be added to the loader's symbol table when that NLM is loaded and removed when the NLM is unloaded.

When the NLM importing a procedure and the NLM, or NetWare kernel, exporting the procedure are running in the same protection domain, a direct procedure call is made to invoke the procedure.

When an NLM contains a call to a procedure in an NLM that is loaded in a different protection domain, an intra-machine remote procedure call (RPC) [BIRR84] is made to invoke the procedure.

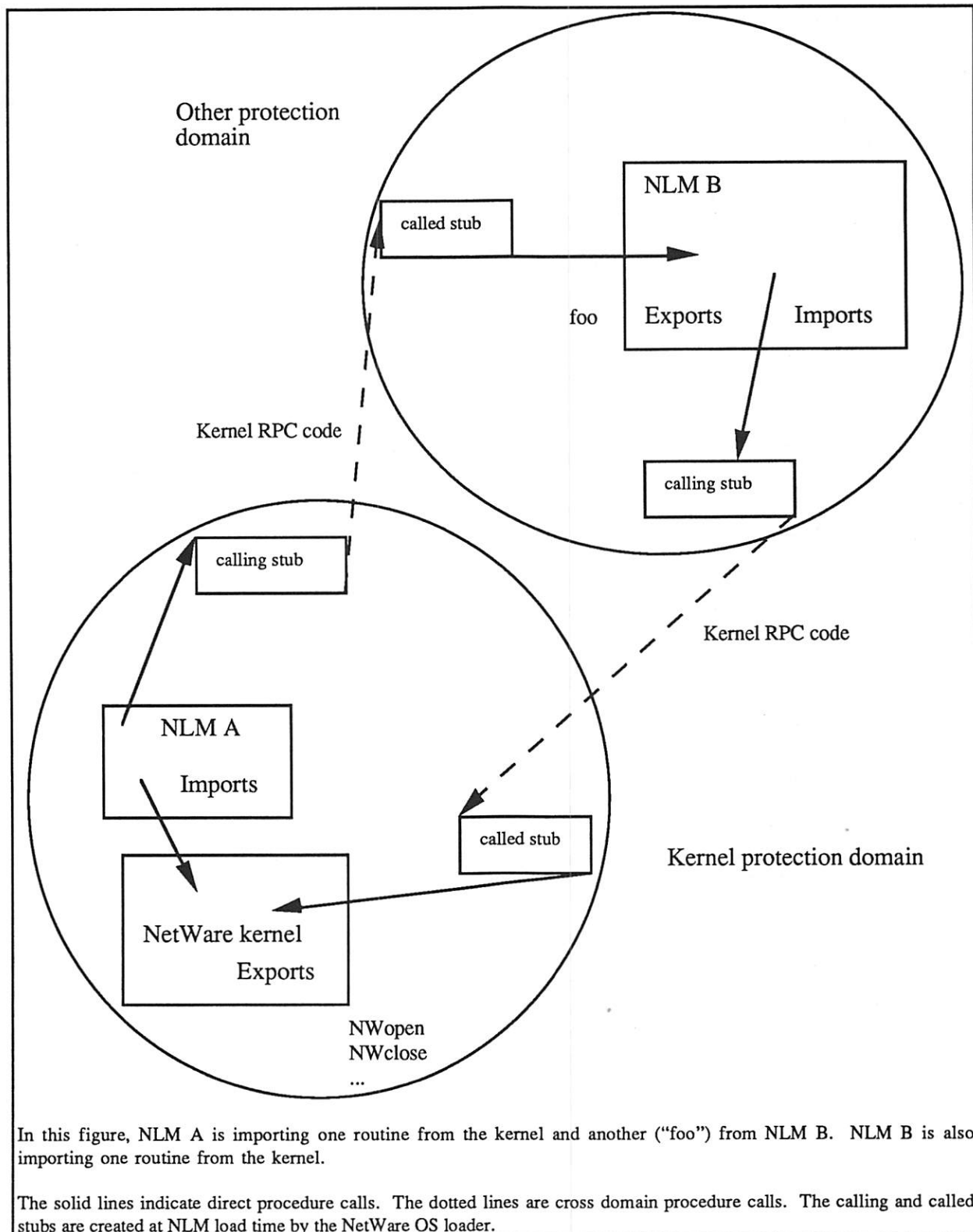
For example, assume that NLM “A” is loaded into a different protection domain than NLM “B”, but NLM “A” contains a call to routine “foo” in NLM “B” (see the following figure).

In a traditional implementation of RPC, a calling stub (user stub in [BIRR84]) linked into NLM A marshals the parameters of the call to foo, and then calls into the kernel to transfer the parameters and control to B. A called stub (server stub in [BIRR84]) linked into NLM B unmarshals the parameters and invokes foo. After foo returns, the called stub marshals any results and then calls into the kernel to return the results and control to the calling stub in A. The calling stub unmarshals the results and returns them to the call point in NLM A that originally called foo.

The NetWare implementation of RPC differs from the traditional RPC model in several respects. The major difference is that the calling and called stubs for exported procedures are automatically generated at load time by the NetWare kernel; they are not linked into either NLM. This isolates developers from knowledge of which procedure calls are direct and which are remote, and allows for increased flexibility in configuring the system. Second, the transfer of parameters and results is done by the kernel, not the calling or called stubs.³ This process is driven by pseudo-code which describes the types of all the parameters to a given procedure. This gives the kernel the ability to make certain optimizations based on the types of the parameters and on the location of the protection domains involved. Third, when a pointer to a callback routine is passed as a parameter in an RPC, the necessary calling stub, called stub, and kernel data structures are generated at run time. This allows callback routines to be invoked across protection domains.

Load-time creation of RPC stubs and kernel data structures is possible because exported routines are described in an interface description language that allows a programmer to provide a full description of the parameters to each exported routine. These descriptions are similar to the “function prototypes” of ANSI C [HARB91] but also allow for the specification of other attributes of the parameters (in particular, whether the parameter is an input parameter, an output parameter, or is both an input and an output parameter). The NetWare interface description language is similar in function to that of the Common Object Request Broker Architecture [OMG91].

³The current implementation of the NetWare kernel makes the calling stack shared between the two protection domains and uses it to transfer the parameters to the called protection domain. Pointers in the parameters result in the kernel making the indicated memory shared between the calling and called protection domains.



The choice of which NLMs should share protection domains involves a tradeoff between performance and fault tolerance. The use of direct procedure calls gives the best performance, but exposes each NLM to failures occurring in the other NLM. Loading communicating NLMs in separate protection domains gives good fault isolation but incurs the performance overhead associated with the RPC mechanism. By basing the system on a procedure call interface, with load-time stub generation, the NetWare OS allows this tradeoff to be made comparatively late: at load time. In this sense, the NetWare OS is similar to Lipto as described in [DRUS92].

Exposing other interprocess communication mechanisms such as message passing or shared memory to the programmer forces a decision on the binding of program units to protection domains to be made at design time in order to get good performance. Message-passing systems [ACCE86] suffer performance problems when the communicating entities are in the same protection domain. Shared memory systems [LI89], on the other hand, suffer performance problems when the memory sharing is not provided by the hardware. By basing a design on either of these paradigms, future flexibility in system structure is reduced. Basing the NetWare OS on the procedure call paradigm allows the kernel to make use of direct procedure calls when the two communicating NLMs are in the same protection domain, to make use of shared memory to carry out the RPC when that is available, and, in the future, to use message passing when the two NLMs are not in the same address space.

3.2. The NLM Execution Environment Provided by the Kernel

When first loaded, an NLM is given one thread of control. It can then create as many additional threads as it needs. Alternatively, the NLM can destroy its only thread, which leaves the NLM existing as a shared library [GING89].

Each NLM has access to a virtual screen and keyboard. The kernel switches the physical system screen and keyboard (in unison) between the various virtual screens and keyboards in the system.

In the Unix operating system, file descriptors 0, 1, and 2 refer to standard input, standard output, and standard error output respectively [RITC78]. The NetWare OS, on the other hand, does not assign any special meaning to these file descriptors. However, the C runtime environment in NetWare does set up these descriptors (as well as allow for I/O redirection [RITC78]).

The NetWare OS does not provide a separation between user space and kernel space. All NLMs running on a server have the same security privileges (though they may be loaded in different protection domains), and all NLMs have access to the same set of importable procedures. For example, the initialization code in an NLM may be written using higher level programming abstractions, such as a standard C library [PLAU92], while the more performance-critical portions of the NLM may use programming interfaces more tightly coupled to the NetWare OS.

An NLM has associated with it certain resources such as main storage and open files. There are mechanisms by which an NLM can transfer ownership of these resources to other NLMs. By convention, an NLM is expected to release all resources it owns before exiting, as part of a philosophy of having developers aware of which resources they are using (in order to minimize resource usage with long-running NLMs). However, if an NLM exits without releasing all its resources, the resources it had owned are returned to the system, and the system console makes note of the fact that the NLM had not released all its resources.

As a thread executes, its program counter moves between NLMs. This may involve moving between protection domains, as an NLM calls routines in a protection domain external to it. Threads in the NetWare OS combine the Spring concepts of "thread" and "shuttle" [HAMI93].

The ability of threads in the NetWare OS to "follow" procedure calls into other NLMs and protection domains is similar to facilities in other systems ([JOHN93] [HAMI93] [FORD94] [FINL90]).

If a thread is blocked (because of a blocking call to the file system, say, or awaiting network I/O), it goes to sleep until it is unblocked. When a thread is unblocked, it is put on the run list until scheduled for execution (section 3.3 discusses the scheduler in more detail).

If an NLM running in a non-kernel protection domain terminates abnormally (because, for example, of a memory protection check), its protection domain is "quarantined". Because of the non-preemptive nature of the NetWare OS, exactly one thread is executing in the protection domain when it terminates, and that thread is suspended when the failure occurs.⁴ Threads are not allowed to call into a quarantined domain; if a thread attempts to call into a quarantined domain, this same mechanism is invoked to suspend the calling thread and quarantine its domain. Previous RPC calls from a quarantined domain are allowed to return to the quarantined domain. Non-suspended threads running in a quarantined domain are allowed to call out from the domain as well as return from the domain.

An abnormal termination in the kernel protection domain is treated as a fatal error.

Because the NetWare OS scheduler is non-preemptive, it is up to each NLM to yield control of the CPU frequently in order that the system remain responsive. Thus, if an NLM has sections that execute for long periods of time without blocking, it will be necessary for the NLM to occasionally call a yield function (exported by the NetWare kernel and described in greater detail in section 3.3). A programmer, in deciding how often to call yield, will attempt increase the "signal to noise ratio" in the program, i.e., make sure that the cost of the yield call is amortized over a significant amount of program processing. This is especially true given that the vast majority of yield calls are, effectively, no-ops (because there is no other thread ready to run). The desire to process a long time between yields conflicts with the goal of having a very responsive system. For this reason, it is important to reduce the cost of doing a null yield in order to encourage the programmer to execute them more frequently. To this effect, in the current NetWare OS, the cost of executing a yield is less than 20 machine instructions if there is nothing else in the queue to run (not including the cost of possibly crossing a protection domain boundary). If there is other work to be done, it takes less than 100 instructions before the new thread is running.

In the case where the NLM calling a yield function is not in the kernel protection domain, an intra-machine RPC must be executed. This significantly increases the cost of the yield call. For example, in the current implementation on an Intel 80486 processor, such a call (with no parameters) adds approximately 50 machine instructions of overhead for the call and the same number for the return. Of each group of 50 machine instructions, however, 49 take approximately two machine cycles each; the other instruction takes approximately 100 machine cycles by itself.

In the future, it is possible to provide preemption in the NetWare kernel but to maintain a non-preemptive execution model within a protection domain. This means that a protection domain could be preempted by the kernel and another protection domain scheduled to run. However, any RPC calls into the preempted domain, or returns back into the preempted domain from previous RPC calls to other protection domains, would need to be blocked until the preempted domain has been rescheduled and itself issued a yield. If this extension to the NetWare kernel were to occur, then the yield calls in a given protection domain would be for sharing the CPU with other threads in that protection domain; other protection domains would not be affected by the frequency of yield calls. A side-effect of this would be to increase the speed of a yield call, because it would not need to cross a protection boundary.

Our experience is that the non-preemptive nature of the NetWare OS is both a blessing and a curse.⁵ We end up with a more deterministic system with fewer of the concurrency issues that are so difficult in many systems. For

⁴There are ways in which a thread can arrange to "catch" such a failure, making the failure look like a non-local return. Depending on the mechanism used, the protection domain in which the failure occurred may or may not be quarantined.

⁵For example, the developers of some NLMs have assumed that file system accesses will block often enough to keep the developer from having to explicitly yield the CPU. Because of the fact that the NetWare file system performs so much caching, this assumption of NLMs quite often turns out to be false. In order to allow these NLMs to run but not monopolize the system, the NetWare file system will occasionally, but deterministically, perform a yield for an NLM, even if the file system operation did not need to block.

example, multi-threaded NLMs can avoid locking shared data structures if their use of these data structures does not explicitly yield or call a service that might yield or block.

Additionally, a non-preemptive system, in our opinion, offers significant performance advantages over more traditional preemptive systems. As an example, in the NetWare file system, a read request on a file that has byte range locks set runs through the linked list that makes up the locking structure for the file. In a preemptive operating system, this code section would have to be protected (by disabling interrupts or by locking the linked list). However, in the NetWare OS, the file system code is written with the understanding that there will be no preemption during the processing of the read request. Thus, the code path is both simpler and faster. In a non-preemptive system, the developer has more control of how often his or her program yields control of the system, and can use this control to simplify the program. Additionally, placing yield calls in a program (which can be done late in the program development cycle) has the added benefit of keeping the developer aware of how long the code paths actually are. The developer effort and mindset necessary to program in this environment require a certain amount of education for the developer. While committed to non-preemption, we continue to explore new tools and other means to ease the burden on the NLM developer.

3.3. The Scheduler

The NetWare OS was designed specifically for the purpose of providing service to networked clients. The performance of servicing network requests is very sensitive to the scheduling policy of the server operating system. The NetWare scheduler is therefore tuned to give high performance in this specific environment.

As mentioned above, the NetWare scheduler provides a non-preemptive scheduling paradigm for the execution of threads. A thread that has control of the machine will not have this control taken away from it unless one of the following occurs:

- it has called one of the yield calls (see below)
- it has blocked (awaiting disk I/O, for example)
- it has returned to the scheduler (for threads dispatched to service work objects – see below)

In particular, interrupts (including timer interrupts) do not cause a thread switch until, at least, the next yield call.

In order to be both non-preemptive and responsive, NLMs are explicitly required to offer to relinquish control of the CPU at regular intervals (not to exceed 150 milliseconds). They do this using one of three yield functions:

`ThreadSwitch()` informs the scheduler that the thread is in a position to continue performing useful work if no other thread is ready to run. This is the normal form of yield for threads that are processing but want to allow other work to execute.

`ThreadSwitchWithDelay()` delays the thread until at least a specific number of yields (from other threads) have been processed. This is quite often used by a thread when it is blocked on a spin lock [JONE80] in order to give the thread holding the lock time to release the lock.

`ThreadSwitchLowPriority()` informs the scheduler that this thread has more work to do, but as a background activity.

Having different yield calls allows the scheduler to determine how to treat the yielding thread, i.e., in which scheduling class the thread belongs and the thread's parameters in that class. An alternative structure would have one yield call and separate calls that a thread could make to set its scheduling class. We are trying to keep threads as lightweight (in terms of state) as possible, so we prefer not to keep a per-thread variable detailing the scheduling class. Additionally, if there were such a state variable per thread, each thread would need to manage this state as it moves between different tasks in the system. This is particularly problematic when coupled with the "work objects" of the NetWare OS (see below), in which a thread may be involved over time in many different parts of the system.

Finally, if the scheduling class were part of a thread's state, we would increase the number of branches in the queuing logic, which may "break" the processor's pipeline. As it is, we know at the entry point to the specific yield routine on which queue we are going to place the thread, so the number of branches is reduced.

There are six basic structures the scheduler uses in scheduling the CPU:

The delayed work to do list is an ordered (by *pollcount* – see below) list of routines to call, together with a parameter for each routine. Routines called from this list run at software interrupt time.

The work to do list is a first-in, first-out (FIFO) list of routines to call, together with a parameter for each routine. Routines called from this list run at process time.

The run list consists of those threads that are ready to run at any given time. This is a FIFO list. Threads run from this list run at process time.

The hardware poll list is a list of routines to be called to check on the status of different hardware components of the system. In contrast to all the other lists mentioned here, entries on this list remain on the list when their associated routines are called (i.e., they must be explicitly removed). Routines called from this list run at software interrupt time.

The low priority list consists of threads that are ready to run but have indicated that they should be run after everything else in the system has run. This list is also serviced in FIFO order. Threads run from this list run at process time.

Pollcount is a monotonically increasing sequence number that is incremented by one for every yield call made.

ThreadSwitch() will cause the calling thread to be enqueued on the run list if control is taken away from the calling thread.

ThreadSwitchLowPriority() will cause the calling thread to be enqueued on the low priority list if control is taken away from the calling thread.

ThreadSwitchWithDelay() will cause an entry to be created and enqueued on the delayed work to do list.⁶ This entry will have its *pollcount* value set to the current pollcount incremented by the value of a system parameter (the so-called "handicap" value, nominally set to 50). This entry will point at a routine that takes as a parameter a thread handle and enqueues that thread on the run queue.

Note that with the exception of the delayed work to do list, all the lists used by the scheduler have a constant insertion and removal time, helping make the scheduler efficient. The delayed work to do list has a variable insertion time, but the removal time is still constant.

There are separate calls to place an entry on the work to do list and on the delayed work to do list. Note that the entries on the work to do list run at process time while entries on the delayed work to do list run at software interrupt time and thus have a more restricted execution environment (for example, they cannot block).

The scheduler basically imposes the following priority on processing in the system (from highest to lowest):

interrupts (these run at interrupt time)

entries on the delayed work to do list that have a pollcount less than or equal to the current value of pollcount (these run at software interrupt time)

⁶In fact, the entry for the work to do list is created on the stack, so no memory allocation actually occurs.

entries in the work to do list (these run at process time)

threads on the run list⁷ (these run at process time)

entries on the hardware poll list (these run at software interrupt time)

threads on the low priority list (these run at process time)

The reality is slightly more complex than the above list because, for example, the scheduler does not allow lower priority tasks to become starved because of higher priority activity. For example, threads on the low priority list are guaranteed to be scheduled at least eighteen times per second – the frequency of the system clock.

The delayed work to do list and the hardware poll list are mechanisms that allow us to trade off responsiveness for higher system capacity by setting certain hardware devices (such as network interface boards or serial line adapters) into a mode in which they do not normally interrupt the system, but rather are serviced periodically by system software. The hardware poll list is a regular, background, polling list. The delayed work to do list can be used to speed up polling a given hardware device when hardware or system loading makes this necessary.

Entries on the work to do list (described in more detail in the next section) are usually invoked to begin processing on a newly received request. Since requests frequently complete without yielding or blocking, calling these entries before processing the run list has the effect of favoring new requests over older, slightly longer running requests. If a request blocks or yields the CPU during its processing, it enters the category of “older, slightly longer running” and, thus, will be scheduled behind newly arriving requests.

The choice between preemption and non-preemption in scheduling is basically a trade off and is based in part on the predicted workload of the system. The NetWare scheduler is optimized for a non-CPU intensive workload. The code path for a typical file system request, for example, is short enough that it contains no yield calls (partly because the code paths do not need to deal with concurrency issues). Additionally, fast thread switch times (on the order of 100 machine instructions) help extend the space of those workloads that perform well in the NetWare OS. Certain CPU intensive workloads, with multiple threads contending for the CPU, might perform better in a preemptive environment.

3.3.1. Work Objects

Earlier versions of the NetWare OS had separate pools of threads dedicated to each different service offered by NetWare (e.g., DOS file service, Unix file service, Macintosh file service, DOS print service). Each pool was created by a particular NLM (in general). As described in the preceding section, the scheduler looks for active threads and schedules them to run. A thread is activated, for example, when an incoming network message arrived on a connection associated with the service provided by that thread.

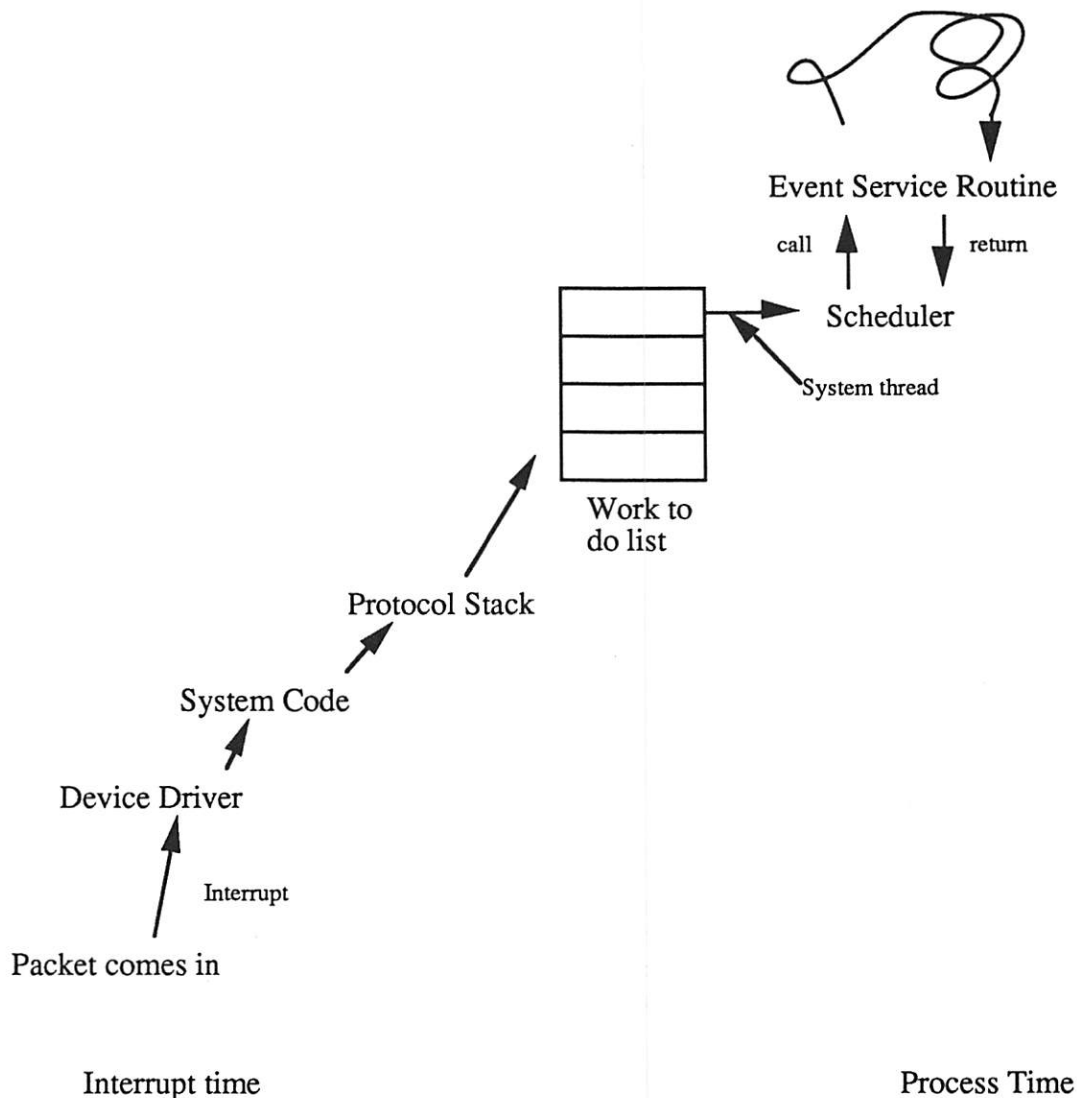
In this environment, each service was responsible for determining how many threads to create, heuristics for when to create new threads or terminate existing threads as the system load changed over time, etc. Implementing these functions, while not difficult, imposed an overhead on the programmer. Additionally, since each thread consumes a certain amount of real memory for such objects as stack space or control blocks, this scheme made inefficient use of memory.

In the most recent version of the NetWare kernel (4.0), a new paradigm for scheduling work has been introduced. This paradigm takes advantage of the fact that NetWare threads carry essentially no state while waiting for a request

⁷Note that threads that have called ThreadSwitchWithDelay() reside on the delayed work to do list but actually have a priority lower than all the threads on the run list. This is because when the entry on the delayed work to do list is actually executed, it will have the effect of enqueueing the thread at the end of the run list.

to service, and so are interchangeable. Thus any thread is often as good as any other thread in performing a service, as long as the call to the service carries the correct parameters.

As mentioned in the previous section, the work to do list consists of a list of tuples, known as “work objects”. Each work object contains the address of a procedure to call, and a parameter to pass to that procedure. As an example of how this works, consider the following figure: a received packet interrupts the system, causing the protocol stack to demultiplex the packet to a given connection. The protocol stack, in the example, builds a work object consisting of the address of an ESR that has been registered for this connection and the address of the incoming packet. The protocol stack then calls into the NetWare scheduler to enqueue the new work object on the work to do list, and then returns from the interrupt.



At process time (during a yield call, or when blocking another thread or because the system was idle when the interrupt occurred) the scheduler examines the work to do list before looking in the run list. In this case, the work to do list is not empty. If the thread running in the scheduler is an otherwise idle system thread (as is often the case), it will call the ESR associated with the first work object directly, with no context switch. Otherwise, a thread will be allocated from a pool of threads managed by the NetWare kernel and will call the ESR. In either case, the ESR will be passed a parameter (in the example, the address of the incoming packet).

The ESR may call other routines, block, and so forth, until it has finished the processing associated with the incoming packet and has returned. The ESR, or any other routine in the processing of the incoming request, may at its option install the extra state that a thread may have.⁸ When the ESR returns, its thread runs the kernel scheduling code, possibly picking up and servicing another entry from the work to do list.

Note that in the above example the transition into the protocol stack could also have been called at process time by using work objects.

Moving away from “per-NLM threads” has allowed the system to be run with a much smaller number of worker (“daemon”) threads, resulting in more efficient use of memory and increased productivity for NLM programmers.

3.4. Paging

The NetWare kernel does not support paging. It is not clear if the primitives supplied by the kernel today would be sufficient to provide paging via a so-called “external pager” [YOUN87]. The page hardware of the underlying system is, however, used to implement protection domains. Currently, the page hardware is also used to keep certain “per-thread” and “per-protection domain” variables at constant virtual addresses.

4. Server Replication

The NetWare OS has long supported fault tolerance approaches such as the mirroring of disk drives. Recently, however, we have developed a new technology, known as “System Fault Tolerance III” (SFT III) that allows for the mirroring of an entire server. In this configuration, two identical server hardware platforms act as one logical server to their clients. At any time, one of the servers acts as the primary and actually replies to requests from the clients. The second server acts as the secondary by tracking the state of the primary and staying ready to perform a failover (in which it takes over from the primary should the primary fail). No special hardware is involved in performing the mirroring and the possible failover.

4.1. Architecture

In order to accomplish this, we have restructured the system into a so-called “I/O engine” and a “mirrored server engine”. The I/O engine contains all code that actually deals with hardware connected to the system, as well as controlling the execution of the mirrored server engine. The mirrored server engine, which actually can run any combination of correctly written, non-hardware-specific NLMs, is the piece that is mirrored between two different systems.

The interface between the I/O and mirrored server engines consists of an “event queue” of events passed from the I/O engine to the mirrored server engine, and a “request queue” moving from the mirrored server engine to the I/O engine. These two queues are the entire interface between the two engines. The mirrored server engine contains a copy of the NetWare kernel complete with scheduler, and so forth, but no code that actually touches any hardware beyond basic CPU registers and the storage allocated to that engine.

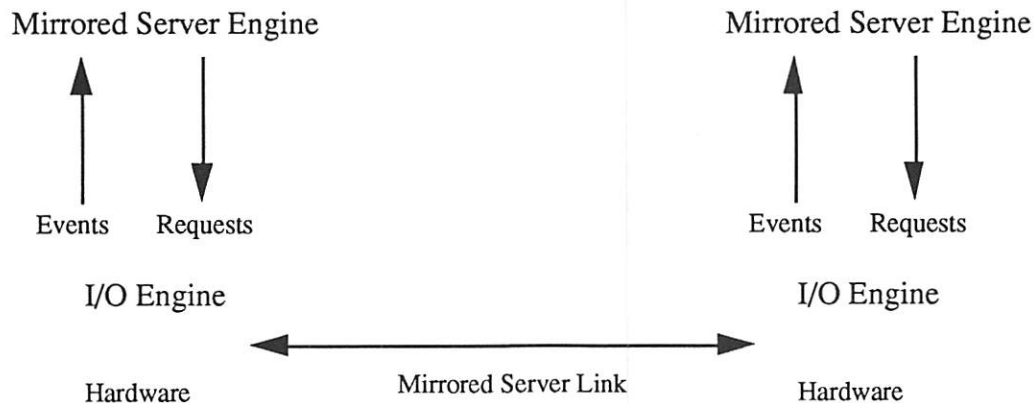
Additionally, the mirrored server engine contains a “virtual” network, with its own network numbers (in the various protocol address spaces), and protocol stacks connected to this virtual network. Note that the network numbers and node numbers on this virtual network are exactly the same for the two mirrored server engines.

The two systems to be mirrored are connected through a high-speed, point-to-point data link running a set of protocols specifically designed to support server replication. The link and the protocols run on the link are known as

⁸This state is related to the C language runtime environment provided by the operating system. Installation of this state takes 24 machine instructions in our Intel 80x86-based implementation.

the “mirrored server link” (MSL). The link itself can be any high-speed data link technology, configured in a point-to-point fashion.⁹ The two systems to be mirrored are also connected via the shared internetwork.

Initially, only one of the systems runs, operating in an “un-mirrored” mode. When the second system comes up, the two systems connect over the MSL. The mirrored server engine on the primary server is suspended momentarily, and its entire memory image is transferred to the secondary, where it is loaded into memory, building an exact copy. Once the memory image and associated state such as register contents exist on both machines, both mirrored server engines are started up. The basic architecture is illustrated in the following diagram.



At the moment of starting, the state in both mirrored server engines is identical. Server mirroring works by keeping the state of the two mirrored server engines identical over time, so that if the primary dies (because of a hardware failure, say), the secondary engine will be able to take over (with almost no action required on the part of the clients¹⁰).

Four basic conditions have made this possible:

- 1) Networks are inherently unreliable, so any packets awaiting transmission at the primary at the time of failover will either be re-requested by the client or retransmitted by the mirrored server engine.
- 2) The mirrored server engine is a non-preemptive system, so the execution path inside the mirrored server engine does not require interrupts for its correct functioning.
- 3) We are able to “virtualize” time, so the actual wall clock time, which will be different at the two mirrored server engines at the same point of execution, is not seen by the mirrored server engines. Thus, the two engines will see the same time at the same points of execution, and thus will operate in exactly the same manner.
- 4) The event queue is the only input the mirrored server engine has from the rest of the world.

Note that hardware physically attached to a system that is down is unavailable. This includes disk drives. To make sure that we are able to continue file service during such an outage, we mirror the disk drives between the two

⁹A 10 megabit/second ethernet link between the two systems is minimally acceptable as an MSL, but for the initial synchronization, as well as to allow for increased distances between the primary and secondary servers, a more specialized, higher speed link is desirable.

¹⁰Some network routing protocols converge slowly in certain topologies. To increase the speed at which the clients start communicating with the old secondary/new primary, a message is sent to the clients causing them to reacquire their route to the mirrored server.

servers. There are other protocols for bringing the mirrored disk drives into synchronization, but those are beyond the scope of this paper.

4.2. Dynamics

Every event enqueued to the primary mirrored server engine is reliably transferred over the MSL to the secondary I/O engine to put on the queue for the secondary mirrored server. No other events are queued up for the secondary mirrored server.

When the mirrored server engine has completed all processing possible on its current event and needs new input, it will request a new event from the I/O engine. The two I/O engines maintain sequence numbers on the requests coming down from the mirrored server engines and cooperate to pass the identical events in response to the same request numbers. Thus, the mirrored server engines see the same set of events, and so behave like any state machines that start off in the same state and are given the same set of input stimuli – they make the same state transitions. In fact, the two mirrored server engines execute the exact same code path between events. The two systems are not in close lock step – one of the mirrored server engines will normally be ahead of the other engine. However, the two engines will be in exactly the same state at the point at which they each make a request of their I/O engine.

The mirrored server engine's notion of time is controlled by the incoming events from the I/O engine. The mirrored server engine does not access the CPU timer directly.

In the normal mode of operation, the secondary I/O engine keeps all requests given to it from the secondary mirrored server engine, but does not act on the majority of these requests.¹¹ The secondary needs to keep track of outstanding requests in order to deal with primary failures. The primary I/O engine executes all requests (with the exception of those directed at the secondary I/O engine). Events generated in the I/O engine as a result of servicing events, or of external events, are collected by the primary I/O engine and communicated to the secondary I/O engine.

As a diagnostic tool we have the capability to capture the request stream generated by the secondary mirrored server engine and compare it with the request stream generated by the primary mirrored server engine. This has been useful in detecting bugs in programs (referencing uninitialized data or data in the process of being updated by the I/O engine).

Additionally, because of our ability to capture the state of a system and capture the event and request streams for that system's subsequent behavior, we have the (currently unrealized) ability to allow customers in the field to record this data when diagnosing a bug and to forward it to their software vendors for replay and analysis.

4.3. Fault Recovery

The primary and secondary I/O engines constantly communicate over the MSL. If the secondary detects that the primary has failed, it will take over for the primary.¹² While up, the network routing protocol running on the primary had been advertising reachability to the virtual network used by the mirrored server engines. Now, the secondary starts advertising this reachability. As soon as the routing topology converges (which depends on the routing protocol in use as well as on the topology between the connected client systems and the two servers), all connections will continue running as if nothing happened.

When the old primary comes back up, it synchronizes its state with the new primary and takes on the role of secondary.

¹¹If the request is, for example, for a disk I/O operation on a disk physically attached to the secondary server, the secondary I/O engine will execute that request.

¹²The secondary, before deciding that the primary is down, attempts to contact the primary across the internetwork. This is done to detect the case where the MSL, and not the primary, has become inoperative.

When applied to file server processes running in the mirrored server engine, this system bears a resemblance to the Harp file system [LISK91]. To our knowledge, however, this is the first system in which a pure software approach to fault tolerance has been applied to such a wide range of services (such as database, mail and messaging, printing, different file service protocols, etc.). Additionally, in SFT III, the knowledge that the system has been replicated is hidden from the specific service processes, being handled by the operating system instead.

The server replication design is described in more detail in [MAJO92].

5. Future Work

As mentioned in section 3.1, currently the system only supports two protection domains. We are working on allowing an arbitrarily large number of protection domains. Currently, portions of the storage allocated to a calling domain are shared with the called domain, a protection exposure which needs to be addressed.

The failure model for protection domains (discussed in section 3.2) will evolve as we get more experience with it.

There is some potential for making use of protection domains in order to run NetWare in a non-shared memory, multiprocessor environment.

Currently, some code paths for interrupt time processing are fairly long. We would like to use work objects to shorten the interrupt time component of these paths. Among other benefits, we expect to get a more robust system by reducing the amount of code that needs to deal with concurrency issues.

Finally, engineering work on NetWare, the product, is ongoing (for example, porting the system to various RISC-based architectures).

6. Acknowledgments

The NetWare kernel and operating system are the result of many people's efforts over the years – too many people to list here. We would like to thank Sam Leffler and the USENIX referees for helpful comments on this paper. Michael Marks provided valuable comments and feedback on earlier drafts of this paper.

7. References

- [ACCE86] Accetta, M. J., W. Baron, R. V. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W. Young, "Mach: A New Kernel Foundation for Unix Development", in *Proceedings of the Summer USENIX Conference*. July, 1986.
- [CLAR85] Clark, David D., "The Structuring of Systems Using Upcalls", in *Proceedings of the 10th Symposium on Operating Systems Principles*. December, 1985.
- [DRUS92] Druschel, Peter, Larry L. Peterson, and Norman C. Hutchinson, "Beyond Micro-Kernel Design: Decoupling Modularity and Protection in Lipto", in *Proceedings of the Twelfth International Conference on Distributed Computing Systems*. June, 1992.
- [FINL90] Finlayson, Ross, Mark D. Hennecke, and Steven L. Goldberg, "Vanguard: A Protocol Suite and OS Kernel for Distributed Object-Oriented Environments", in *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, October, 1990.
- [FORD94] Ford, Bryan, and Jay Lepreau, "Evolving Mach 3.0 to a Migrating Thread Model", in *Proceedings of the Winter USENIX Conference*. January, 1994.
- [GIEN90] Gien, Michel, "Micro-Kernel Design", in *Unix Review*, 8(11):58-63. November, 1990.

- [GING89] Gingell, Robert A., "Shared Libraries", in *Unix Review*, 7(8):56-66. August, 1989.
- [HAMI93] Hamilton, Graham, and Panos Kougiouris, "The Spring Nucleus: A Microkernel for Objects", in *Proceedings of the Summer USENIX Conference*. June, 1993.
- [HARB91] Harbison, Samuel P., and Guy L. Steele Jr., *C, A Reference Manual*. 1991.
- [JOHN93] Johnson, David, and Willy Zwaenepoel, "The Peregrine High-Performance RPC System", *Software - Practice & Experience*, 23(2):201-221. February, 1993.
- [JONE80] Jones, Anita K., and Peter Schwartz, "Experience Using Multiprocessor Systems - A Status Report", in *ACM Computing Surveys*, 12(2):121-165. June, 1980.
- [LEFF89] Leffler, Samuel J., Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*. 1989.
- [LI89] Li, Kai, and Paul Hudak, "Memory Coherence in Shared Virtual Memory Systems", in *ACM Transactions on Computer Systems*, 7(4):321-359. November, 1989.
- [LISK91] Liskov, Barbara, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams, "Replication in the Harp File System", in *Proceedings of the 13th Symposium on Operating Systems Principles*. December, 1991.
- [MAJO92] Major, Drew, Kyle Powell, and Dale Neibaur, "Fault Tolerant Computer System", in *United States Patent 5,157,663*. October, 1992.
- [OMG91] Object Management Group, *The Common Object Request Broker: Architecture and Specification*. 1991.
- [RITC78] Ritchie, D. M., and K. Thompson, "The UNIX Time-sharing System", in *Bell System Technical Journal*, 57(6):1905-1929. July-August, 1978.
- [TANE92] Tanenbaum, Andrew S., *Modern Operating Systems*. 1992.
- [YOUN87] Young, M., A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System", in *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*. November, 1987.

Author Information

Drew Major is chief scientist and systems architect at Novell, working with the software development team. He was instrumental in the design and implementation of the NetWare operating system, scheduler, file system, and server replication. He holds a B.S. degree in Computer Science from Brigham Young University.

Greg Minshall is involved in the design of networking and operating systems at Novell. He holds an A.B. degree in Pure Mathematics from the University of California at Berkeley. His e-mail address is minshall@wc.novell.com.

Kyle Powell is a senior systems architect at Novell, having been heavily involved in the design and implementation of the client portion of NetWare, as well as in server replication and the operating system itself. He holds a B.S. degree in Computer Science from Brigham Young University.

NetWare is a trademark of Novell, Inc.; UNIX is a registered trademark of UNIX System Laboratories, Inc., a subsidiary of Novell, Inc.; Macintosh is a registered trademark of Apple Computer, Inc.; all other product names mentioned herein are the trademarks of their respective owners.

THE USENIX ASSOCIATION

The USENIX Association is a not-for-profit membership organization of those individuals and institutions with an interest in UNIX and UNIX-like systems and, by extension, C++, X windows, and other programming tools. It is dedicated to:

- * sharing ideas and experience relevant to UNIX or UNIX inspired and advanced computing systems,
- * fostering innovation and communicating both research and technological developments,
- * providing a neutral forum for the exercise of critical thought and airing of technical issues.

Founded in 1975, USENIX is well known for its twice-a-year technical conferences, accompanied by tutorial programs and vendor displays. Also sponsored are frequent single-topic conferences and symposia. USENIX publishes proceedings of its meetings, the bi-monthly newsletter *;login:*, the refereed technical quarterly, *Computing Systems*, and has expanded its publishing role in cooperation with the MIT Press with a book series on advanced computing systems. The Association actively participates in various ANSI, IEEE and ISO standards efforts with a paid representative attending selected meetings. News of standards efforts and reports of many meetings are reported in *;login:*.

SAGE, the System Administrators Guild

The System Administrators Guild (SAGE) is a Special Technical Group within the USENIX Association, devoted to the furtherance of the profession of system administration. SAGE brings together system administrators for professional development, for the sharing of problems and solutions, and to provide a common voice to users, management, and vendors on topics of system administration.

A number of working groups within SAGE are focusing on special topics such as conferences, local organizations, professional and technical standards, policies, system and network security, publications, and education. USENIX and SAGE will work jointly to publish technical information and sponsor conferences, tutorials, and local groups in the systems administration field.

To become a SAGE member you must be a member of USENIX as well. There are six classes of membership in the USENIX Association, differentiated primarily by the fees paid and services provided.

USENIX Association membership services include:

- * Subscription to *;login:*, a bi-monthly newsletter;
- * Subscription to *Computing Systems*, a refereed technical quarterly;
- * Discounts on various UNIX and technical publications available for purchase;
- * Discounts on registration fees to twice-a-year technical conferences and tutorial programs and to the periodic single-topic symposia;
- * The right to vote on matters affecting the Association, its bylaws, election of its directors and officers;
- * The right to join Special Technical Groups such as SAGE.

Supporting Members of the USENIX Association:

ANDATACO	OTA Limited Partnership
ASANTÉ Technologies, Inc.	Quality Micro Systems, Inc.
Frame Technology Corporation	Sybase, Inc.
Matsushita Electrical Industrial Co., Ltd.	UUNET Technologies, Inc.
Network Computing Devices, Inc	

For further information about membership, conferences or publications, contact:

The USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

Email: office@usenix.org
Phone: +1-510-528-8649
Fax: +1-510-548-5738

ISBN 1-880446-58-8